Pyrrha: Congestion-Root-Based Flow Control to Eliminate Head-of-Line Blocking in Datacenter

Kexin Liu*[*], *Zhaochen Zhang**[*], *Chang Liu**, *Yizhi Wang**, *Vamsi Addanki*$^{\triangle}$, *Stefan Schmid*$^{\triangle}$,
Qingyue Wang*, *Wei Chen**, *Xiaoliang Wang**, *Jiaqi Zheng**, *Wenhao Sun*$^{+}$, *Tao Wu*$^{+}$, *Ke Meng*$^{+}$,
Fei Chen^{+}, *Weiguang Wang*$^{+}$, *Bingyang Liu*$^{+}$, *Wanchun Dou**, *Guihai Chen**, *Chen Tian**
**Nanjing University,* $^{\triangle}$*TU Berlin,*$^{+}$*Huawei, China*

Abstract

In modern datacenters, the effectiveness of end-to-end congestion control (CC) is quickly diminishing with the rapid bandwidth evolution. Per-hop flow control (FC) can react to congestion more promptly. However, a coarse-grained FC can result in Head-Of-Line (HOL) blocking. A fine-grained, per-flow FC can eliminate HOL blocking caused by flow control, however, it does not scale well.

This paper presents Pyrrha, a scalable flow control approach that provably eliminates HOL blocking while using a minimum number of queues. In Pyrrha, flow control first takes effect on the root of the congestion, i.e., the port where congestion occurs. And then flows are controlled according to their contributed congestion roots. A prototype of Pyrrha is implemented on Tofino2 switches. Compared with state-of-the-art approaches, the average FCT of uncongested flows is reduced by 42%-98%, and 99th-tail latency can be 1.6×-215× lower, without compromising the performance of congested flows.

1 Introduction

Given the increasingly stringent performance requirements on datacenter networks, avoiding congestion and the resulting delays has become critical for many applications. Indeed, measurement studies show that congestion events are frequent in today's datacenters, *e.g.*, bursty key-value stores [18, 30], web search services with massive queries [11], and data-parallel/machine-learning systems with partition/aggregation traffic patterns [29, 90, 92, 13, 26, 91, 4, 70, 33]. Generally, congestion occurs at an output port when the arrival rate of traffic exceeds its link bandwidth. Queues build up at congested ports. With an inflated buffer, flows could endure a long queuing delay or even face packet loss, hence flows' completion times (FCT) can be prolonged [11, 56].

State-of-the-art approaches to handle congestion is end-to-end congestion control (CC) [11, 95, 65, 56, 19, 55, 61, 15, 35, 6, 14, 86, 58]. Congestion can be detected by senders when congestion signals are sent back or feedback delay is observed. Usually, it costs senders at least one Round-Trip Time (RTT) to be aware of the congestion, and then senders may take several RTTs to converge to an appropriate transmission rate. In modern datacenters, the effectiveness of end-to-end CC is quickly diminishing with the rapid bandwidth evolution [36, 43, 74, 22](§ 2.1).

With the increasing link speed and performance requirements of datacenter networks, an intriguing and emerging alternative is per-hop flow control (FC), which can react to congestion much more promptly. It suppresses the transmission of the upstream entity before overwhelming the downstream queue, which avoids a large buffer occupancy. Generally, traffic in the same queue is controlled as a whole. Once the queue length exceeds a given threshold, a pause frame can be sent to pause the upstream entity [16, 17], avoiding further buffer build-up where congestion occurs. However, a coarse-grained flow control might spread the congestion to the whole network, inducing Head-Of-Line (HOL) blocking, and hurting the performance of victim flows [40, 54, 62, 57]. Here HOL blocking refers to flows being paused innocently (§ 2.2). A naïve approach to eliminate HOL blocking could be to isolate each flow into different queues and control each of them separately. However, such a per-flow granularity flow control is not scalable since the hardware resources of switches are limited. State-of-the-art flow control approaches hence aim at reducing the number of queues required by compromising the granularity of isolation [36]. Thus, the HOL blocking can not be eliminated entirely (§ 2.3).

This paper explores how to eliminate HOL blocking in a scalable manner, *i.e.*, minimizing the required number of queues. We observe that when congestion occurs, flow control first takes effect on the root of the congestion, *i.e.*, the port where congestion occurs. Then several congestion hotspots (*i.e.*, output port with buffer build-up) can appear along the back pressure path of flow control. These involved hotspots form a congestion tree, where the root of the

[*] First author and second author contribute equally to this work.

tree is the root of the congestion. Simply controlling the transmission of flows based on hotspots could induce HOL blocking. Instead, if we separately control the transmission of flows according to the congestion root they participate in, flows will not be paused innocently. As the number of concurrent congestion roots observed on each port is moderate, only a reasonably small number of queues on each port are required for traffic isolation in each switch [20].

Based on these insights, this paper presents Pyrrha, a congestion-root-based per-hop flow control protocol (§ 3). A framework is constructed for analyzing egress-based flow control algorithms. We analytically prove that Pyrrha is a HOL-blocking-free per-hop flow control protocol with the minimum queue requirements.

In Pyrrha, each switch maintains a snapshot of the congestion status of the downstream network. Flows that would pass through the same congestion root in their downstream paths could be pushed into a dedicated congestion queue locally in each upstream switch. Then, flows passing through different congestion roots can be controlled separately as soon as possible. Pyrrha solves a set of challenges.

- ***How to identify a hotspot's role?*** When congestion occurs, multiple congestion roots may be claimed in succession. This could occur when an upstream congestion hotspot claims itself root while its downstream hotspot disagrees, and vice versa. With a local view, it is hard to tell which root is the exact root in the upcoming congestion. Intuitively, given the behavior of flow control, congestion roots always locate on the most downstream ports that flows pass through. Inspired by the root-selection procedure of classic spanning tree algorithms [7, 32], Pyrrha employs a distributed self-stabilizing *merge* mechanism. A port can claim itself a *self-nominated* congestion root independently when it detects its queue buildup for the first time. It then abdicates its claim in favor of a *self-nominated* downstream root if (part of) its flows pass through the downstream claimer. Quickly, participating hotspots can converge to a congestion tree. Naturally, the congestion root is detected. (§ 4.1).
- ***How to identify a congested flow upon its arrival?*** For each arriving flow to a switch, its entire following path should be deterministic to the switch to identify a congested flow. Inspired by recent industrial path control practice [73, 52], Pyrrha proposes a hash-function-aware design for switches. Every switch can determine the path that a flow will take. With that information, the switch could match the path against the congestion status snapshot of the downstream network to determine whether it is a congested flow (§ 4.2).
- ***How to handle events-tangling scenarios?*** Congestion trees could overlap with each other which could result in a congested flow traversing several congestion roots. Besides, the congestion root in networks may vary with transient bursty traffic. Without careful scheduling among congestion queues, such flows could be mistakenly paused/resumed or delivered out-of-order. Pyrrha proposes a resource-efficient hierarchical queue structure corresponding to the physical topology. The design ensures both correct flow control semantic and in-order delivery even in highly dynamic scenarios (§ 4.3).

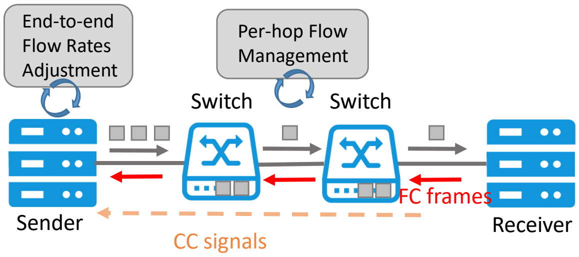


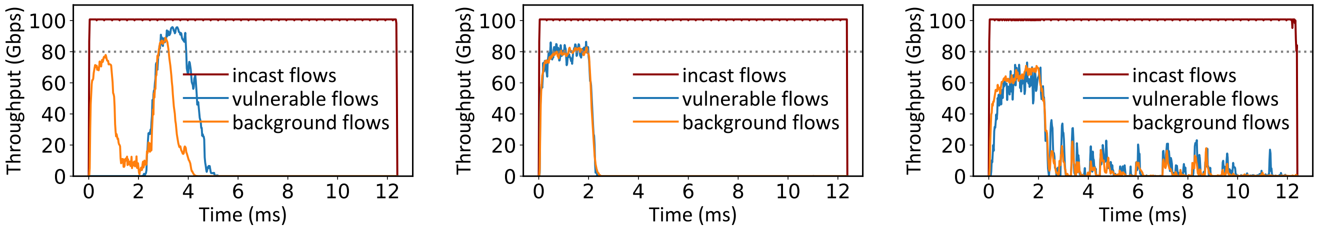
Figure 1: Our vision of labor division between CC/FC.

A prototype of Pyrrha is implemented on Tofino2 [48]. Testbed evaluations and large-scale NS-3 simulations have been performed. We compare Pyrrha with existing flow control protocols (*e.g.*, Priority Flow Control (PFC) [46], and BFC [36]). And we also incorporate Pyrrha with existing congestion control protocols (*e.g.*, DCQCN [95], TIMELY [65], and HPCC [56]). We find that the average FCT of uncongested flows is reduced by 42.8%-98.2%, and 99th-tail latency can be 1.6×-215× lower, without compromising the performance of congested flows. In addition, Pyrrha reduces the maximum buffer occupancy by up to 1.8×-6.2× (§ 6). As a contribution to the research community and to ensure reproducibility, our artefacts will be made publicly available online together with this paper. *This work does not raise any ethical issues.*

2 Background and Motivation

2.1 CC is Falling and FC is Rising

A variety of datacenter applications produce bursty traffic, which can result in different types of congestion, *e.g.*, incast, and load imbalance. To handle congestion, existing efforts focus on developing end-to-end congestion control (CC). CC can be classified into *reactive* and *proactive*. With reactive CC, congestion can be detected by switches (*e.g.*, ECN in DCTCP [11] and DCQCN [95], INT measurements in HPCC [56], PINT [19], PowerTCP [6] and Poseidon [86]) or end-hosts (*e.g.*, Timely [65], Swift [55], and On-Ramp [61]). After receiving congestion signals or if packet delays are observed, senders adjust the transmission rate. It may cost a flow several RTTs to converge to an appropriate rate even in a stable network condition. With proactive CC, bandwidth is allocated before the transmission (*e.g.*, ExpressPass [25, 58], Homa [67], NDP [41], Aeolus [43], and pHost [31]). However, whether to transmit in the first RTT is a dilemma, and proactive CC either wastes the first RTT or risks reintroducing congestion. Recently there are CCs [14, 69] which detect congestion at sub-RTT by

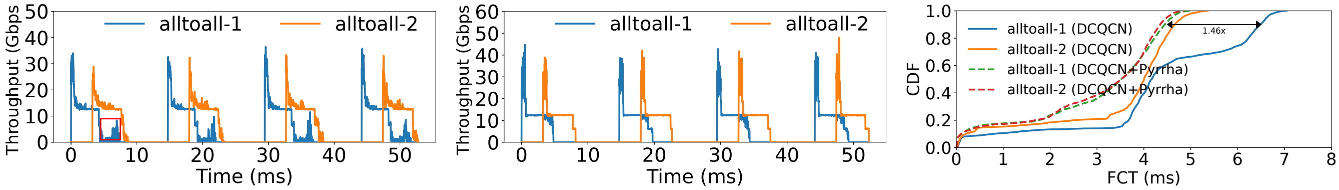


(a) PFC.

(b) per-flow queue flow control.

(c) BFC.

Figure 2: Performance when incast flows are mixed with non-incast flows.



(a) Throughput (DCQCN).

(b) Throughput (DCQCN+Pyrrha).

(c) FCT comparison.

Figure 3: Performance comparison under MoE workloads (collided phase).

leveraging switches to send back control frames directly. However, they can not quickly react to congestion especially when congestion occurs at the last hop (*e.g.*, incast).

Several trends. The control loop of CC is too long to handle transient congestion, given the fast evolution of datacenter networks: (i) The high port bandwidth allows to send out more flows within the first RTT, even before congestion control could step in [36, 43]. Transient bursty traffic results in a large buffer occupancy and at the same time mislead the rate adjustment of CC. (ii) The buffer size cannot catch up with the increased speed of its high bandwidth per port [36, 22]. It becomes harder for switches to buffer transient congestion and wait for end-to-end CC's intervention. (iii) The growing scale of datacenter networks and the emerging workloads (*e.g.*, distributed training) lead to more bursty traffic (*e.g.*, a larger scale incast) [74, 42, 75].

Our vision. To handle bursty traffic, per-hop FC protocols should step in. As shown in Figure 1, we propose a labor division between CC and FC:

- **(i) Per-hop FC handles transient congestion.** A switch can control the traffic transmission quickly by per-hop flow control frames. It is in a unique position to quickly manage flows that have already been injected into the network to avoid performance downgrades.
- **(ii) CC takes its part to handle persistent congestion.** CC can adjust the flow rates to increase/decrease the forthcoming traffic injection into the network when congestion occurs and to handle fairness issues.

2.2 HOL Blocking Problem of Simple FC

In RoCEv2 [16], PFC [46] ensures that the buffer does not overflow. PFC pauses the upstream entity at a per-port or per-priority-class queue granularity when the ingress queue length exceeds a given threshold. Further, when the upstream ingress queue exceeds the threshold, a pause frame will be sent to its upstream entities.

However, the intervention of PFC could spread congestion. When PFC is applied, flows that do not contribute to downstream congestion could be paused when they share the same queue with congested flows. The congestion scope can spread from congested ports to piles of innocent ports. Hence, it could downgrade the performance of uncongested flows. We define the above congestion spreading phenomenon as the **HOL blocking caused by flow control**, *i.e.*, a flow is paused innocently by the congested port that it does not pass through. HOL blocking could cause a throughput downgrade. More severely, PFC is vulnerable to deadlock with routing loops [40, 54, 62, 57].

Typical incast workloads. To demonstrate the HOL blocking problem, we conduct a simulation where incast flows are mixed with non-incast flows. 720-to-1 incast flows are generated with an average size of four Bandwidth-Delay Product (BDP) and non-incast flows are generated with a load of 0.8 following the Poisson arrival process (setting details in § 6). Figure 2(a) depicts the real-time throughput. To make it more clear, we use ***vulnerable flows*** to denote uncongested flows sharing paths with congested flows in the remainder of the paper since they are more likely to be hurt by congested flows. Other uncongested flows are denoted as ***background flows***. Hence, the throughput of vulnerable flows is severely hurt since they are paused by downstream switches with congested flows as a whole, leading to a large queuing delay. Besides, since a PFC pause frame storm occurs, congestion is spread to the whole network. Consequently, background flows suffer a throughput downgrade from 1ms to 4ms.

MoE workloads. We investigate the performance of Pyrrha under the traffic of a popular type of pre-trained large language model called Mixture-of-Expert (MoE). According to [42], the traffic pattern can be characterized by an imbalanced all-to-all where a significant portion of the traffic is sent to a few 'hot' experts. Owning to the synchronization nature of the training process, the traffic exhibits a periodic on-off pattern [75, 76]. Following [75], two groups of periodic traffic are generated.

Figure 3 shows the performance of flows with collided phase, *i.e.*, where their phase overlaps (the results of the interleaved phase are detailed in Appendix B). In the case of DCQCN+PFC, two groups of all-to-all suffer from HOL-blocking as they compete for bandwidth, which in turn triggers PFC. The throughput of the all-to-all1 group is notably suppressed, even dropping to zero upon the arrival of the all-to-all2 group. Once a portion of the all-to-all-2 flows finishes, all-to-all1 begins to grasp some of the bandwidth, as indicated by the red rectangle in the figure. While for DCQCN+Pyrrha, benefiting from the rapid reaction to the congestion, two groups of flows do not disturb each other, accelerating the tail latency by a factor of 1.46×.

2.3 State-of-the-art Flow Control is Flawed

To overcome the HOL blocking problem caused by coarse-grained control on queues, a naïve scheme is per-flow queue FC scheme. Figure 2(b) demonstrates the simulation results under the same settings as in Figure 2(a) when the switch assigns a dedicated queue to each flow passing through it. Vulnerable and background flows fully utilize the link, at the same time the throughput of incast flows does not downgrade (see [59] for theoretical analysis). However, tens of thousands of flows can be observed on ports [84]. Hence a per-flow granularity scheme is non-scalable.

Existing flow control approaches try to reduce the number of queues by compromising the isolation granularity.

Destination-based flow control. This line of work tries to isolate congestion by separating flows transmitting to different destinations. Revisiting super-computing literature decades ago, per-destination Virtual Output Queues (VOQs) [23, 27, 51] are assigned to separate flows with different destination addresses [85, 68]. However, per-destination VOQs are not scalable since the number of VOQs required scales with the number of hosts in networks. Floodgate [60] is a per-hop flow control leveraging per-destination windows to identify incast traffic in datacenter networks. Then, incast traffic can be isolated from non-incast traffic. However, it should maintain a per-destination state of the remaining sending window which demands much memory resources on switches. In summary, they only aim at eliminating HOL blocking caused by the last-hop incast and cannot handle other types of congestion such as load imbalance. In addition, they require per-destination resources which build barriers to deployment at scale.

Queue-based flow control. A second line of work targets at assigning flows into a limited number of queues to alleviate HOL blocking. Since it cannot isolate congested flows from uncongested flows entirely, HOL blocking cannot be avoided. In BFC [36], flows are assigned to a number of queues (*i.e.*, 32-128 queues per port) according to their *flow-id*s and hash functions. A flow is assigned to an empty queue if possible and could share it with other flows when all queues are occupied. As shown in Figure 2(c), with relatively large incast flows, both vulnerable and background flows maintain a very low throughput from 2ms to 12ms. The tail latency of flows is prolonged by 6× compared with the per-flow queue scheme. This is because incast flows can occupy queues for a long time. Vulnerable and background flows sharing the same queue with incast flows are severely hurt, and their transmission rate is mistakenly controlled by the network bottleneck (*i.e.*, the destination ToR of incast).

To sum up, existing solutions cannot totally avoid HOL-blocking, and some of them are impractical.

3 Pyrrha Overview

Our target is to eliminate HOL blocking caused by flow control in the most cost-effective way. In this section, we first illustrate why congestion root is the appropriate granularity of flow control along with the basic idea of Pyrrha, followed by a list of challenges.

3.1 Basic Idea

Before illustrating our basic idea, we introduce several concepts in flow control when congestion occurs. Figure 4 shows a typical congestion tree rooted at P5.

- **Congestion Hotspot.** When the congestion occurs, a flow control scheme starts to pause upstream, inducing several hotspots along its back-pressuring path. A congestion hotspot is an output port whose input rate exceeds its output rate and its queue accumulates.
- **Congestion Root.** As its name implies, a congestion root is the root cause of the congestion, where congested flows finally aggregate. Meanwhile, it is the root of the corresponding congestion tree.
- **Congestion Tree.** A congestion tree can be made up of a root (*e.g.*, P5), non-root hotspots (*e.g.*, P1-P3) and leaf ports (already controlled by the root but have not paused its upstream yet). In our paper, a congestion tree is named after its root (*e.g.*, T5 denotes the tree whose root is P5).

Why congestion-root-based FC? A non-differentiating treatment of flows passing through congestion roots and hotspots could result in HOL-blocking since flows passing through hotspots might not contribute to the congestion. To avoid involving innocent flows, flow control should decide the right scope of flows to control. Intuitions are that if a flow control only applies pause to flows contributing to the congestion root, HOL-blocking can be eliminated.

Note that the peak bandwidth lasts for approximately 1 ms during which non-hot experts complete their traffic reception, followed by the hot experts continue receiving their traffic.

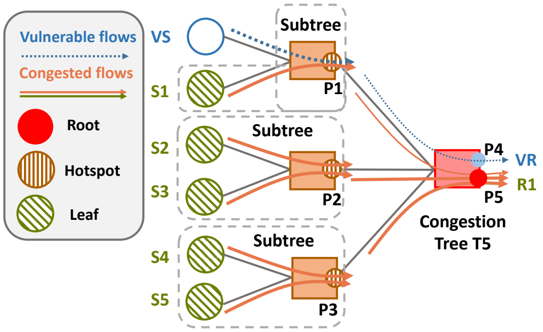


Figure 4: Congestion tree illustration.

Meanwhile, flows passing through different congestion roots should be handled separately to avoid interfering with others.

Quick reaction to congestion: identify congested flows upon arrival. Once a congestion root is detected, the congestion root information is propagated to its upstream switches when it is detected. Then each switch maintains a congestion status snapshot of its downstream networks. A congested flow can be detected upon its arrival in networks by checking whether its path matches existing congestion roots. Hence, its transmission can be controlled several hops earlier before it arrives at congestion roots. It reduces the occurrence of severe congestion and relieves the buffer pressure on following hops, especially on congestion roots.

Fine-grained isolation: manage traffic according to its contributed congestion roots. By default, flows are pushed into a physical output queue (OQ). For separate control, congested flows passing through the same congestion root should be pushed into a dedicated isolation queue (IQ) assigned to the corresponding congestion root. Different categories of flows, *e.g.*, congested flows contributing to different congestion roots, and uncongested flows are isolated respectively. Then, the transmission of congested flows can be controlled precisely by pausing the exact queue assigned to the congestion root, which avoids congestion spreading.

Based on the above ideas, we propose Pyrrha, a practical fine-grained flow control scheme based on congestion roots. Pyrrha achieves good properties, as shown next.

Theorem 3.1. *Pyrrha is a HOL-blocking free per-hop flow control protocol requiring the minimal number of queues.*

Proof. Inspired by the gradient graph [78, 77], we represent flows and queues as vertices, and their traversing relationship as edges. Congested flows contributing to the congested queue's congestion can be represented as its immediate neighbors. Based on this representation, congested flows can be identified as the neighbors of a congested queue. Similarly, potentially paused flows by a congested queue can be identified in our graph representation. In order to prove that Pyrrha is HOL-blocking-free, we formally prove that the set of congested flows and the set of potentially paused flows are identical for any congested queue. Futhermore, we prove that any flow control using fewer queues than Pyrrha cannot avoid HOL blocking. We use Dirichlet's box principle and proof by contradiction to establish this impossibility result. Our formal proof appears in [59]. □

3.2 Design Challenges

Identify congestion roots. Correctly identifying the congestion root is the prerequisite of proper flow control. In tree T5 of Figure 4, P1-P3, and P5 should agree that P5 is the root of incast. Otherwise, if P1 incorrectly identifies itself as an independent root, vulnerable flow VS→VR becomes a victim.

Identify congested flows. To control the transmission of flows separately, congested flows should be identified precisely. When an uncongested flow is mistakenly identified (*e.g.*, VS→VR) as a congested flow, it can be paused innocently. To recognize a congested flow quickly, a switch should obtain the flow's path to be traversed in its downstream network so that it can check the path against its downstream network snapshot.

Handle tree-tangling scenarios. Several concurrent congestion trees can be intertwined among themselves. If non-root ports of trees are overlapped, these ports can play different roles in different congestion trees. These ports should not entangle transmission control received from different trees. When a tree is covered by another one, it results in a flow contributing to several congestion roots, which appear frequently in real systems. One may wonder why congestion still occurs at a port after part of its passing flows is controlled by its downstream congestion roots. This is because the congestion at the port is caused by itself other than its downstream ports. Figure 5(b) depicts an example that the path of a flow can first match a congestion root (P6) and then match another congestion root (P5). The transmission of the congested flow should be controlled by both congestion roots, *i.e.*, only when both congestion roots send a RESUME frame can the flow be transmitted. Besides, the transmission control of the flow should not interfere other flows sharing one of the congestion roots.

In addition, when the competing traffic changes, the congestion tree can shrink or expand, leading to congestion roots varying over time. For P3 in Figure 4, initially, packets of flows {S4-S5}→R1 are controlled in the IQ_{P5}. If flows {S1-S3}→R1 finish, P3 is likely to become the new congestion root later. Flows {S4-S5}→R1 are no longer the congested flows of P5, instead, they are congested flows of P3. Later-arrived packets of these flows should be carefully scheduled in case they are transmitted before packets previously queued into the IQ_{P5}, which results in out-of-order delivery.

4 Pyrrha Design

Figure 5 demonstrates the architecture of Pyrrha. The bottom part of the figure denotes the packet propagation among Pyrrha switches. And the upper part depicts the three major components of a Pyrrha switch. Congestion

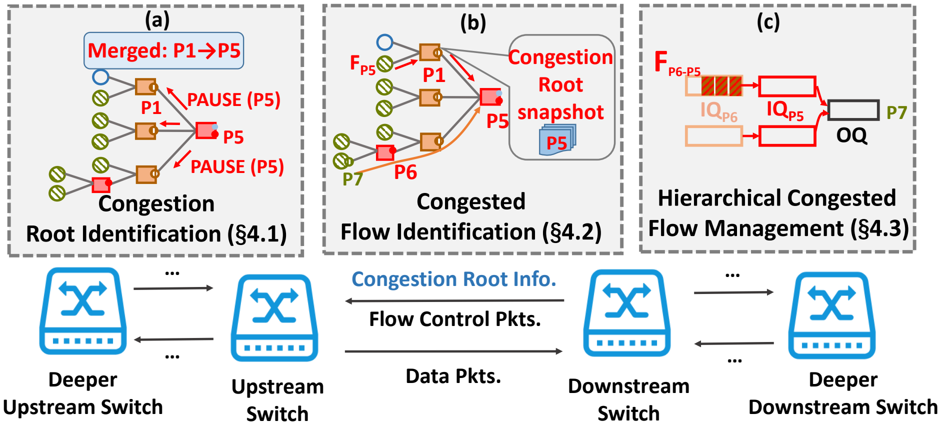


Figure 5: Pyrrha architecture.

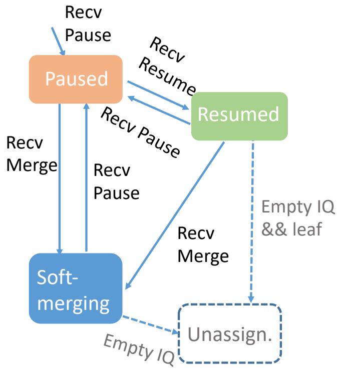


Figure 6: IQ state transitions.

Root Identification (§ 4.1) responds to downstream switches to detect congestion and identify the corresponding root. The congestion information is carried in flow control frames (*e.g.*, PAUSE) and propagated to upstream switches in a hop-by-hop manner. Congested Flow Identification (§ 4.2) maintains the snapshot of downstream network congestion states to help quickly recognize congested flows through path matching. Isolation queues (IQs) are structured in a hierarchy corresponding to the topology by Congested Flow Management (§ 4.3). Besides, details and discussions are put in § 4.4 and § 4.5.

4.1 Congestion Root Identification

Initial detection. Inspired by the root-election process of spanning tree protocols, a congested port can claim itself a *self-nominated* congestion root candidate independently. Initially, each port is attached with a OQ and flows are pushed into the OQ by default. Hence the queue length increase on the OQ can be regarded as an indication of congestion. When a data packet arrives at the OQ, a switch checks whether the queue length exceeds a given threshold K_{pause} (*e.g.*, several per-hop BDPs). If so, a hotspot is detected and the hotspot regards itself as a congestion root. Subsequent packets that arrive at the hotspot trigger a PAUSE frame to the corresponding upstream port from which the packet arrived.

Congestion root identification. According to behavior of flow control, a root is always the most downstream hotspot in a tree. Hence, we can identify the real congestion roots by merging upstream congestion tree into a more downstream one. As shown in Figure 5(a), when a congestion-root-candidate hotspot (P1) receives a PAUSE frame from a downstream root (P5), it indicates that part of its passing-through flows also traverses this downstream hotspot. Hence it recognizes itself as a false-positive congestion root. A new IQ for this new congestion root (P5) is assigned. All following packets matching the new root will enter the corresponding IQ. Then, the old congestion tree is canceled and merged into the new congestion tree. Note that this process can be iterative when there exist multiple layers of hotspots in a congestion tree. The false-positive congestion roots are eventually merged and the root of the new congestion tree is the real congestion root.

Merging process. To start merging, the false-positive congestion root notifies all its child nodes by sending a control message MERGE. MERGE is sent to all its upstream entities belonging to the (old) false-positive congestion root, carrying the ID of both old and new congestion roots. As shown in Figure 6, switches receiving the MERGE frame change the state of the corresponding IQ to soft-merging and propagate the notification to its upstream further. Soft-merging means that the old IQ now belongs to no congestion tree and can be unassigned once empty. The packets queuing in the old IQ are not controlled by the false-positive congestion root. Instead, only packets passing through the real congestion root are controlled (§ 4.3). The merging process finishes within a one-way delay, hence false-positive congestion roots have a negligible impact on performance.

4.2 Congested Flow Identification

Intuitions are that a congested flow passes through at least one congestion root.

Determining a flow's exact path. To determine whether a data packet belongs to a congested flow, the entire onward-path of each arriving packet at a switch should be deterministic. Pyrrha is compatible with traffic load balancing protocols that can locally get deterministic onward paths for flows [1, 44, 79, 64, 52]. Among those load balancing protocols, hash-based protocols (*e.g.*, per-flow ECMP and PLB [73]) are most widely deployed given its no-reordering properties [87]. Pyrrha proposes a hash-function-aware design. Pyrrha's switch calculates a packet's onward-path by using its IP tuples, routing hash functions, and seeds of its downstream switches, together with flow labels carried in its header if necessary as input. Besides, source routing is compatible with Pyrrha naturally since Pyrrha's switch can derive the onward-path of a packet by parsing its header.

The memory and computation resource to get a flow's path is moderate, and following optimizations are facilitated by leveraging the industry-standard practices in datacenters. (i) Given that switches in datacenter only support limited types of hash functions (*e.g.*, CRC or XOR) to conduct efficient calculation [87, 93, 72], Pyrrha switch only needs to store the type of the hash function of other switches along with their hash seeds. (ii) In many widely deployed topologies [8, 37, 62, 82], the multiple equal path property and the up-down routing strategy can be leveraged to optimize the overhead. In those topologies, the forwarding tables of all core switches are identical. Hence, Pyrrha switch only needs to store one replica for redundant forwarding tables. Furthermore, for fat-tree, the path from a core switch to a given destination is unique, hence Pyrrha switch only conducts hash function calculation for the first two hops.

For other adaptive load balancing approaches, *e.g.*, per-packet spraying and DRILL [67, 34], which determine the path of flows through dynamic states, Pyrrha is a complementary solution to handle the hotspots caused by destination collision (*i.e.*, incast), where load balancing falls short. Experiments in Appendix 6.3 show that Pyrrha is compatible with DRILL and further improves the tail latency by 18.3% compared to pure Pyrrha. Pyrrha also handles corner cases where flows are re-routed to different paths due to link failures (Appendix § A).

Matching and maintaining snapshot. Pyrrha's switch maintains the *congestion-root table* of the downstream networks. As shown in Figure 5(b), the table maintains a snapshot of the congestion states of its downstream networks. When a PAUSE frame indicating a new congestion root is received, the congestion root is recorded in the table. When a packet arrives at the switch, its path can be obtained via above mentioned methods. The switch checks whether its path matches any entry in the congestion-root table. For port P1 in Figure 5(b), packets that will traverse P5 in its downstream path are identified as belonging to a congested flow. Hence, it's enqueued to a separate IQ, which is paused/resumed based on the state of the corresponding root. Otherwise, the packet belongs to an uncongested flow and is put in the OQ.

4.3 Congested Flow Management

To handle tangled scenarios where congestion trees are intertwined among each other or congestion roots vary over time, Pyrrha leverages a hierarchical methodology to manage congested flows corresponding to that of the topology. It can be supported by a Hierarchical Isolation Queue (HIQ) architecture, which manages congested flows in a hierarchy. Pyrrha installs HIQ during compilation according to its location in networks and manages the usage of queues dynamically during runtime through a mapping table. We also provide single-tier IQs prototype to fully support the function of HIQ.

Handling congested flow in hierarchy. Congestion trees are intertwined when non-root ports of trees are overlapped or a tree is covered by another one. For the first scenario, IQs on ports can naturally isolate control from different congestion roots on non-root ports. For the latter one, a congested flow could pass through multiple congestion roots. To ensure precise control isolation, a congested flow should match all corresponding IQs before it is forwarded. A hierarchical organization of IQs based on the location of their corresponding congestion roots in the topology enables a congested flow to match appropriate IQs *sequentially*. Especially, when per-flow load balancing is used, a flow at most encounters one congestion root among switches at the same level. It ensures in-order delivery when a congested flow alters its matched IQs.

Hierarchical Isolation Queue (HIQ) architecture. HIQ consists of several levels of IQs. Each IQ is positioned in a hierarchy according to its distance to the corresponding congestion root in the physical topology. Hence, the number of layers of the HIQ is determined by its location in the network. As shown in Figure 5(c), in a two-tier network, an uplink port of a ToR switch maintains two levels of IQs, since the farthest potential congestion root is two hops from it. Figure 5(c) depicts the HIQ architecture on P7, *i.e.*, the leaf port in Figure 5(b). P7 fully utilizes the two-level architecture of the HIQ, since the farthest congestion root P5 is two hops from it. Especially, the OQ is connected to the last level of the HIQ architecture. A dedicated scheduler is equipped for each level of queues to schedule the traffic transmission. Only when an IQ is in a resumed or soft-merging state can its packet be dequeued and pushed into the next-level IQs/OQ. From the perspective of a packet, this process is performed iteratively until it reaches the OQ. In the scenario depicted in Figure 5, when a flow that will traverse congestion root P6 and P5 arrives at the upstream switch of port P6, it matches the HIQ *from-near-to-far*. After a packet is dequeued from IQ$_{P6}$, it is pushed into a next-level IQ$_{P5}$.When there is no more matched IQ, it is pushed into the OQ. Hence, the packet can be forwarded to the next hop only after all matched congested roots are resumed. In this way, congested flows are controlled locally precisely.

IQs in HIQ are arranged by levels, supporting in-order delivery naturally. Considering the merging procedure in Figure 5(a), the congestion root is changed from P1 to P5. IQ$_{P1}$ is in the soft-merging state and packets in it can be mixed with congested and uncongested traffic. Pyrrha should handle them separately to avoid HOL blocking. Congested flows of root P5 are dequeued from IQ$_{P1}$ and then pushed into the next level IQ$_{P5}$. Uncongested flows are forwarded to OQ. In this way, precise isolation is achieved without inducing re-ordering.

Handling secession of the congestion root. Once the OQ length of the congestion root decreases below the resume threshold, it sends back RESUME to its upstream. Likewise, upstream switches could resume their upstream when their own IQ decreases below the resume threshold. It is an iterative process that the congestion tree eliminates starting from the leaf switches to the root. A congestion port becomes a leaf when it has not sent PAUSE yet, or when the status of all its upstream entities is set to *unassigned*. The IQ of the leaf switch is unassigned when it becomes empty. When all the upstream IQs of a congestion root are marked unassigned and the queue length of OQ of the congestion root is below the resume threshold, the congestion root disappears naturally. Figure 6 depicts the state transition of the IQ usage. Only when an empty IQ is in a resumed or soft-merging state can it be marked as unassigned.

4.4 Miscellaneous Detailed Design

Congestion information propagation. There are three types of flow control frames in Pyrrha, *e.g.*, PAUSE, RESUME, and MERGE. These control frames carry the congestion information and control the transmission of congested flows accurately in a hop-by-hop manner. Correspondingly, there are three states for an IQ, *i.e.*, paused, resumed, or soft-merging. The state transitions of an IQ are shown in Figure 6.

When a congestion root is detected, a PAUSE frame is sent back to the upstream port through the ingress port of which the data packet is just received. Once a data packet is pushed into the OQ which is attached to the congestion root, Pyrrha switch checks the packet's ingress port and sends back the PAUSE frame. A PAUSE frame carries the ID of the congestion root (*i.e.*, identified as *switch-id:port-id*). Likewise, when the queue length of IQ exceeds the threshold K_{pause}, a PAUSE frame carrying the root ID is sent back to its upstream switch.

Cooperation on end-hosts. To handle persistent congestion, end-hosts should control the upcoming traffic into networks. Pyrrha can cooperate with congestion control protocols. And Pyrrha can perform better if end-hosts can respond to PAUSE (or RESUME and MERGE) frames. To pause and resume at a per-flow granularity, end-hosts could leverage a pull-based transmission model, which can be implemented by programmable smart NICs in RDMA networks. Especially, Pyrrha can also handle end-host congestion (*e.g.*, PCIe congestion) by backpressuring the traffic it receives.

Handling rare packet loss. Pyrrha reduces queuing length significantly. Hence, buffer overflow rarely occurs. However, Pyrrha does not guarantee lossless for rare cases where every port of a switch becomes the victim port of a k-1:1 incast lasting for one-hop RTT in turn, where k denotes the number of ports on switches. In this certain case, Pyrrha can start to drop packet and leverage IRN [66] for fast retransmission.

4.5 Scalability Discussions

Queue consumption. In the most extreme scenario, the number of congestion roots can be the number of ToRs in the network. However, the concurrent amount of congestion in networks is usually moderate. It is reported that only 3% of the links in edge and aggregation layers appear as a hotspot for more than 0.1% of time intervals [20, 21]. Moreover, the concurrent roots can be much less than that of hotspots.

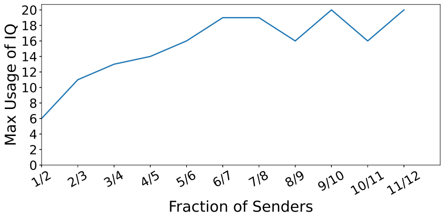


Figure 7: Queue usage analysis.

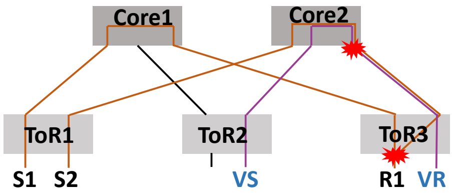


Figure 8: Testbed topology.

To investigate the IQ usage, evaluations under stressful workloads where $(m-1)$ out of m ToRs send traffic simultaneously to the left 1/m of the ToRs are conducted. A k=12 fat-tree topology is employed and each host sends 40 one-BDP flows continuously to create a substantial network burst. Figure 7 illustrates the IQ usage of Pyrrha as the fraction of senders varies. The IQ utilization ranges from 6-20, approximately proportional to the number of pods k, which considered to be relatively moderate. Since commodity switches can support thousands of VOQs [23, 27, 51], assigning a dedicated queue to each downstream root is feasible. To handle corner cases where IQs are not enough, similar to BFC, Pyrrha leverages hash functions [39, 71] to choose an IQ according to the congestion root ID, at a cost of sacrificing precise isolation.

Deadlock prevention. Pyrrha is deadlock robust since OQs never get paused. To prevent cyclic buffer dependencies (CBD) caused by routing loops, Pyrrha switch checks whether the congestion root carried in the PAUSE frame is identical to its own identifier. If so, it ignores the PAUSE frame directly. (see detailed discussions in [59].)

5 Implementation and Testbed Experiments

We implement a Pyrrha prototype on Tofino2, a state-of-the-art programmable switch ASIC [48] with Reconfigurable Match Table (RMT) architecture. In this section, we briefly describe the key modules of the prototype, followed by the overhead analysis. Testbed evaluations show that Pyrrha can achieve good performance (§ 5.2). More implementation details are deferred to Appendix D and E.

5.1 Prototype of Pyrrha

We implement a Pyrrha prototype on Tofino2 with 2.5k lines of P4 code and 2k lines of Python code. The operations of Pyrrha is implemented entirely in the data plane at line rate.

Key modules and the pipeline. The Pyrrha prototype is mainly composed of several modules, *i.e.*, (i) congestion root matcher, (ii) queue manager, (iii) queue state detector, and (iv) signal packet module.

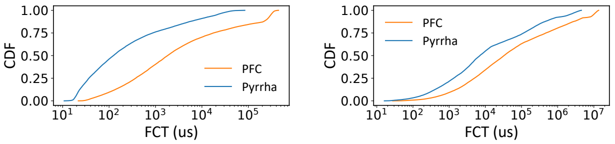
(i) Upon arrival, the data packet first undergoes a standard processing procedure, including forwarding and admission control. Subsequently, the data packet is forwarded to the congestion root matcher, an integral component of the path calculation unit and a congestion root table, facilitating the congested flow identification. Specifically, the path calculation unit calculates the packet's egress ports of its onward path. The congestion root table records the status of ports, indicating whether the port is congested or not. (ii) Then packets enter the queue manager for queue assignment. The queue manager assigns queues to traffic based on the congestion roots it would pass through. Central to its design is a multi-segment stack, wherein each segment manages the available queues for a specific egress port.

(iii) The queue state detector checks whether the length of the assigned queue exceeds the pause threshold or decreases below the resume threshold, and then triggers appropriate signal packets. The queue length is retrieved by utilizing *ghost threads* in Tofino2.

(iv) When it is necessary to send a signal packet, the signal packet module leverages the packet trigger functionality to construct signal packets, such as PAUSE and RESUME. Upon receiving a PAUSE or RESUME, the module engages Tofino2's *AFC (Advanced Flow Control)* mechanism to pause (or resume) the queue.

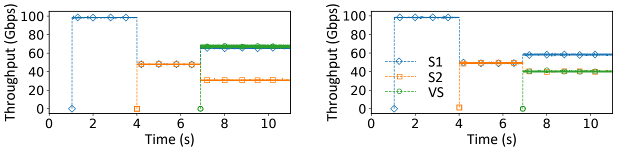
Feasibility of HIQ The architecture of HIQ is supported in current Metro Ethernet (MetroE) service routers [3, 2]. And a recent work of implementing multi-level scheduler on ASIC [94] also verifies its feasibility. According to private talks with chip vendors, they consider it possible to implement HIQ in their next-generation switching chips. For instance, the two-layer HIQ can be obtained by connecting two traffic manager models in series and specifying the next-level IQ to be pushed in when a packet is dequeued. Although HIQ is not supported by the architecture of Tofino2 currently, the features of HIQ can be fully supported via single-tier queues (Appendix C).

Complexity and overhead. Tofino2 adopts pipeline architecture, wherein the resource allocation is determined at compile time. It enables us to ascertain Pyrrha's resource requirements without running it in a large-scale cluster. According to the statistics reported in megascale [49], the scale of current data centers can reach up to 10,000 hosts. Therefore, we use a k=36 fat-tree topology with 11,664 hosts as a representative case. Pyrrha prototype can easily scale to it, with around 11 MB (*i.e.*, 44.5% of Tofino2) of the memory resource consumption. Specifically, the memory usage of Pyrrha prototype is mainly composed of three units, *i.e.*, path calculation unit, congestion root table, and queue manager, overall consuming 9.25 MB. And the processing logic consumes around 1.88 MB. (i) As analyzed in § 4.2, the memory consumption of the path calculation can be optimized by leveraging the industry standard, occupying 0.44MB SRAM. (ii) The congestion root table is organized hierarchically, where the nth table records ports that are n hop away from the switch, and the port is identified as <switch-id, port-id>. Hence, the storage usage of the congestion root table is proportional to the number of ports in networks, occupying 176 KB SRAM. (iii) The queue manager firstly checks whether the congested flow is assigned with a queue through a *IsAssigned Table* and assigns a queue if necessary by looking up a multi-segment *QueueId stack* which records the available queue. Then it updates the *QueueId record table* to record the queue assignment status. This module overall consumes 8.64MB SRAM. Furthermore, leveraging the rapidly maturating HIQ technology [94], Pyrrha can scale to topologies that is an order of magnitude larger. (detailed analysis are deferred to Appendix E).



(a) Vulnerable flows (Web Server). (b) Vulnerable flows (Web Search).

Figure 9: FCT of testbed experiments.



(a) Pyrrha (b) PFC

Figure 10: Throughput of testbed experiments.

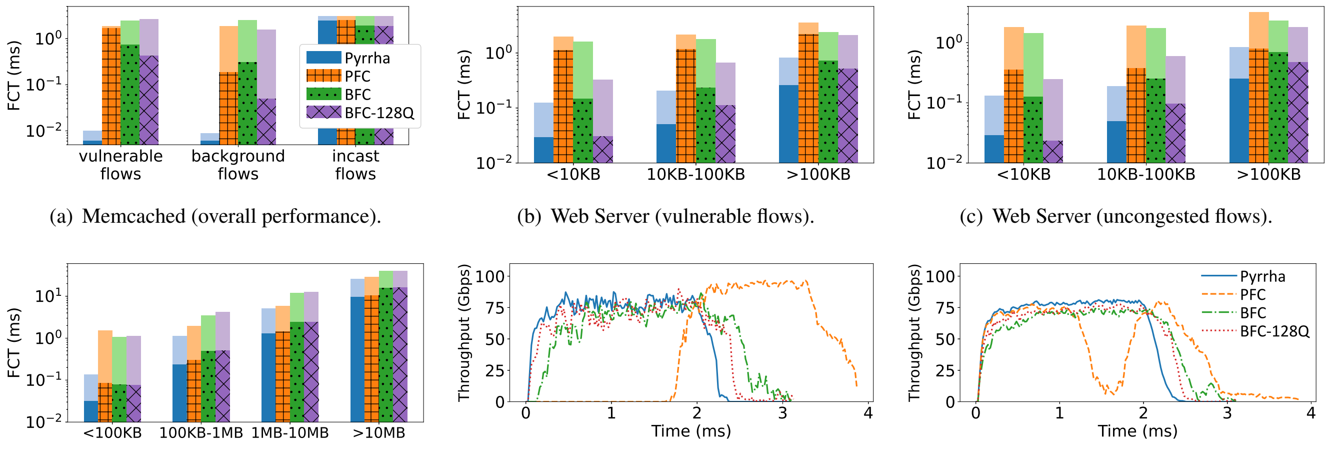
5.2 Testbed Evaluation

Topology. We use a 2-level leaf-spine topology as shown in Figure 8, consisting of three ToR, two core switches, and two hosts per rack, all connected via 100 Gbps links.

Workloads. We evaluate Pyrrha under incast-mix scenarios. Incast flows are generated by letting hosts S1 and S2 transmit flows to host R1 simultaneously. Vulnerable flows are generated by letting host VS send flows to host VR. Flows S2→R1 and VS→VR share the same port on the core switch. Web Server and Web Search flows are generated following a Poisson arrival process (§ 6).

Pyrrha reduces the FCT of vulnerable flows. Figure 9 demonstrates the FCT performance of vulnerable flows. For PFC, vulnerable flows are HOL blocked by incast flows, suffering a large queuing delay. Pyrrha quickly detects the congestion on the destination ToR switch and isolates incast flows into a dedicated queue. Thus, the FCT of vulnerable flows is greatly reduced.

Pyrrha improves the throughput. Figure 10(a) shows the throughput when flows on three hosts S1, S2, and VS start to arrive at 1s, 4s, and 7s, respectively. Pyrrha improves



(a) Memcached (overall performance).

(b) Web Server (vulnerable flows).

(c) Web Server (uncongested flows).

(d) Web Search (uncongested flows).

(e) Web Server (vulnerable flows).

(f) Web Server (background flows).

Figure 11: Performance of FC. The deep/light color in the figures represents the average/99th-tail value for each bar.

the throughput of vulnerable flows to 66.7 Gbps without compromising the overall throughput of incast flows (100 Gbps). The network throughput is improved by 26.7 Gbps.

6 Simulation Evaluation

Topology. A non-blocking clos-network is used. It contains 4 core switches, 10 ToRs, and 160 hosts, similar to the topology used in [67]). Each ToR is connected to its hosts and cores via 100/400 Gbps links, respectively. The per-hop propagation delay is 600ns. The base RTT is 5.1μs, and the base BDP is 64KB. A 3-tier fat-tree topology with 1024 hosts is also leveraged to investigate the scalability of Pyrrha.

Workloads. Under incastmix scenarios, flows following a Poisson arrival process with a load of 0.8 and periodic incast flows each composed of 30-40 MTUs with a load of 0.5 are generated. An incast destination does not receive Poisson arrival flows hence the traffic load does not exceed link bandwidth. The incast degree is 720-to-1. For Poisson arrival flows, three workloads are used [67, 80, 11], where Memcached is composed of small flows, where most of the flows are smaller than 1KB, and Web Server and Web Search are large flows mixed with small flows where a small ratio of large flows dominate the average flow size.

Parameters. There are two parameters of Pyrrha, *i.e.*, the threshold to pause (*i.e.*, K_{pause} times one-hop BDP) and the threshold to resume (*i.e.*, K_{resume} times one-hop BDP). In our evaluations, K_{pause} is set to 2, and K_{resume} is set to 1. Besides, the maximum number of IQs can be used is set to 100, but Pyrrha only uses a dozen of IQs in most cases. A dedicated subsection discusses why these values are used (§B.1). The switch buffer capacity is 20MB. Pyrrha uses shared buffer mode. PFC uses dynamic threshold and $\alpha = 2$.

Metrics. Average/99th-tail FCTs are evaluated. We monitor maximum buffer on each hop to investigate the composition of buffer reallocation that Pyrrha brings.

6.1 Comparing with Flow Control

In this section, we use large-scale NS3 simulations to compare Pyrrha [59] with existing flow control protocols (*e.g.*, PFC and BFC). For BFC, two versions with 32 and 128 queues per port are used, as in its paper (*e.g.*, BFC is used to denote BFC-32). Figure 11 shows the FCT and throughput.

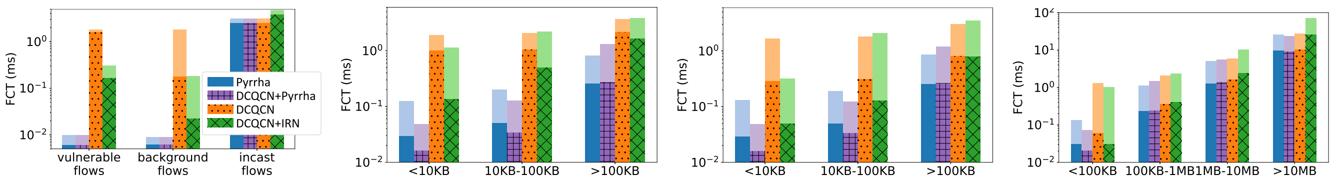
6.1.1 PFC

PFC hurts the performance of uncongested flows since they can be paused innocently by their downstream ports when incast occurs. This especially hurts the performance of workloads that are composed of small flows (*e.g.*, Memcached). For Web Server workload, PFC even spreads the congestion to the whole network thus background flows that do not share the same destination pods of incast flows also get hurt. The throughput performance depicted in Figure 11(f) is a side note to this issue. For background flows, it achieves stable high throughput until PFC pause frame storm occurs at 1.5ms. Then background flows endure a significant throughput loss that lasts for about milliseconds.

6.1.2 BFC

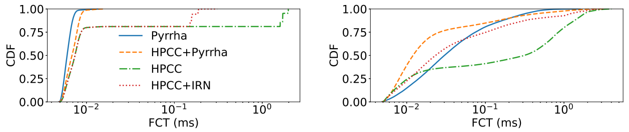
BFC assigns flows to multiple queues according to flow identifiers (FID) and hash functions. It can partially alleviate HOL blocking caused by congested flows and improve the performance of uncongested flows to an extent compared to PFC. However, HOL blocking occurs when congested flows and uncongested flows share the same queue, or flows are hashed into the same flow FIDs. Hence, BFC can not obtain extremely low latency as Pyrrha does. Along with the number of queues used by BFC increases, *i.e.*, from 32 to 128, the performance of BFC is improved. For Web Search workload, the tail latency of BFC is not good because BFC sets a relatively smaller threshold to detect congestion compared to PFC. It can risk spreading congestion.

The third group of bars in Figure 11(a) shows the performance of incast (congested) flows. More results of incast flows are deferred to Appendix B. Pyrrha does not



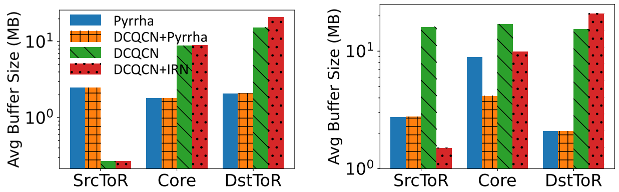
(a) Memcached (overall flows). (b) Web Server (vulnerable flows). (c) Web Server (uncongested flows). (d) Web Search (uncongested flows).

Figure 12: FCT performance (DCQCN).



(a) HPCC: Memcached (overall flows). (b) HPCC: Web Server (uncongested flows).

Figure 13: FCT performance.



(a) Memcached. (b) Web Server.

Figure 14: Buffer occupancy (DCQCN) is split into three parts, *i.e.*, queue length on SrcToR, core and DstToR.

compromise the performance of incast flows compared to PFC. BFC reduces the average FCT of incast flows because BFC splits different flows into different queues and incast flows may use several queues simultaneously. Incast flows can use more bandwidth during the resume phase via the round-robin scheduling mechanism among queues.

6.2 Cooperating with Congestion Control

In this section, we incorporate Pyrrha with state-of-the-art CC (*e.g.*, DCQCN and HPCC). Experiment results of TIMELY is left in Appendix B. The authors' contributed simulation codes, if available, are used in our evaluations [10, 53]. Following [56, 61], a per-flow sending window on hosts is added to all approaches, limiting the in-flight packets of a flow. PFC is used to provide a lossless network for CC by default. Besides, to fully understand the performance of CC, we also evaluate pure CC and use IRN [66] to handle packet loss. IRN leverages selective retransmission to perform an efficient loss recovery.

6.2.1 DCQCN

Since Pyrrha reduces the buffer occupancy by a large extent, we set $K_{min} = 50KB$ and $K_{max} = 200KB$ to cooperate with DCQCN. Figure 12 depicts the average and 99th-tail FCT across different workloads. DCQCN alone can not handle incast flows quickly thus PFC is triggered. Particularly, under Web Server, a PFC pause frame storm occurs. PFC is triggered from the destination ToR of incast and finally reaches the hosts that the source ToR switches connected to, which affects background flows.

Given that PFC has some side effects, one may wonder how DCQCN performs when leveraging IRN for retransmission instead of relying on PFC. However, IRN is not a cure for DCQCN. As shown in Figure 26(b), although DCQCN with IRN reduces the FCTs of small flows compared to DCQCN with PFC, FCTs of larger flows are prolonged. Packet loss hurts the goodput of networks. Besides, when all inflight packets are lost, loss recovery can only get triggered via timeouts. Timeout retransmission hurts the performance of small flows and the tail latency of large flows.

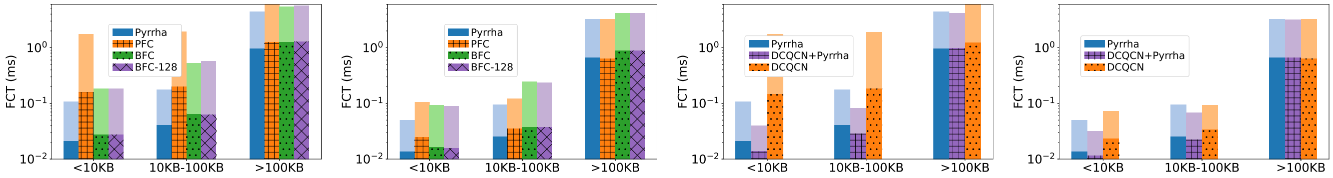
When adding Pyrrha to DCQCN, Pyrrha can significantly benefit uncongested flows (*e.g.*, vulnerable and background flows) by quickly recognizing and isolating incast flows. Pyrrha reacts quickly on in-network congestion by handling it locally, which is especially beneficial to small flows. Therefore, Pyrrha's performance improvement on small flows is the most significant. When the flow size increases, the improvement becomes relatively less obvious. Because large flows give more time for DCQCN to take effect.

Figure 14 shows the maximum buffer occupancy among different hops. Instead of letting congested flows overflow the congestion root, Pyrrha can quickly control their transmission locally. Then the buffer occupancy on the congestion root, *i.e.*, the destination ToR switch, can be greatly reduced. Since Pyrrha pushes back congested flows, the buffer occupancy on the source ToR switch is increased.

From another perspective, when adding DCQCN to Pyrrha, the FCT of small flows can be improved further compared to pure Pyrrha. This is because DCQCN can control the transmission of flows based on a finer granularity than a pause-resume manner. For flows sharing the same path, DCQCN can adjust the rate of each flow individually while Pyrrha controls them as a whole. And congestion control favors small flows since large flows are more likely to be marked and convey the congestion signals back to senders. In addition, the buffer occupancy on the core switch can be further reduced, as shown in Figure 14(b).

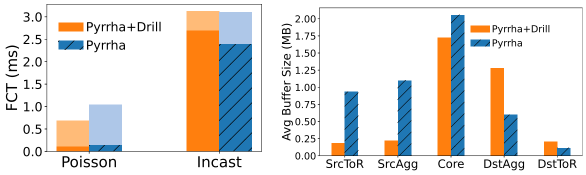
6.2.2 HPCC

When adding Pyrrha to HPCC, uncongested flows carry the queue length of the OQ to avoid unnecessary rate reduction on them. And congested flows carry the queue length of



(a) Web Server (vulnerable flows). (b) Web Server (uncongested flows). (c) Web Server (vulnerable flows). (d) Web Server (uncongested flows).

Figure 15: FCT performance under k=16 fat-tree.



(a) FCT performance. (b) Buffer occcupancy.

Figure 16: Cooperating Pyrrha with DRILL.

IQs. For clarity, Figure 13 shows the FCT distribution of HPCC. Pyrrha helps improve the performance of HPCC. While HPCC achieves good performance on a portion of small flows, as shown in Figure 26(g), this is because HPCC leverages INT to get the in-network information to reduce the queue length aggressively, which benefits small flows. As a cost, its tail latency is prolonged since the transient small flows could interfere with the rate adjustment process and influence the network throughput.

6.3 Additional evaluations and Discussions

Performance under Large-scale Topology. To investigate the scalability of Pyrrha, evaluations under k=16 fat-tree topology (*i.e.*, 1024 hosts) is conducted. Figure 15 shows the performance of different protocols under Web Server. The trend of performance differences between different algorithms is very similar to that of the results in § 6. We analyze the results from two perspectives: (i) When comparing FC with CC protocols, especially for flows smaller than 100KB, fine-grained FC protocols such as BFC and Pyrrha outperform PFC as well as CC protocols. This advantage is attributed to the rapid response of fine-grained FC to network congestion, which provides a degree of mitigation against HOL blocking. (ii) In terms of cooperation between FC and CC, as depicted in Figure 15(c), when Pyrrha is integrated with CC, small flows can achieve the lowest latency. This is because Pyrrha rapidly reacts to the network congestion, preventing HOL blocking for traffic that has already been injected into the network. Concurrently, CC address the persistent congestion associated with larger flows.

Cooperating with adaptive load balancing scheme. Load balancing schemes handle congestion by re-arranging the traffic to avoid path collision. However, they can not handle congestion caused by destination collision (*i.e.*, incast), and Pyrrha acts as a complementary solution to handle it. Pyrrha works together with adaptive routing by only recognizing and handling the congestion of incast.

We investigate the performance when Pyrrha works with DRILL, an adaptive load scheme that chooses the forwarding port according to the queue length dynamically. Similar to § 6, incast-mix traffic is generated and a k=8 fat-tree topology is used. Results in Figure 16 show that Pyrrha works well with DRILL. The integration of DRILL with Pyrrha further reduces the latency of Poisson flows compared to pure Pyrrha. The performance gain is attributed to DRILL's ability to distribute traffic effectively, thereby preventing queue buildup at the upstream (*e.g.*, SrcToR, SrcAgg, and Core), as shown in Figure 16(b). However, this optimization comes with a trade-off in the latency of incast flows, which is higher compared to the pure Pyrrha.

Additional evaluations. In Appendix B, we further explore the performance of Pyrrha under several scenarios, (i) when the size of incast flows varies, (ii) when facing multiple congestion roots, (iii) MoE traffic with interleaved phase, and (iv) when the congestion roots vary. We also compare Pyrrha with per-flow queue scheme and CC without sending window. Besides, the parameter selection and merging mechanism of Pyrrha is evaluated. A variant of Pyrrha using single-tier queues is investigated (§ C.2).

Discussions. Discussions regarding how Pyrrha handles link failures, as well as a review of related works, are put in the Appendix A.

7 Conclusion

This paper was motivated for a labor division between congestion control and flow control. It is time to embrace per-hop flow control in datacenter networks to react to congestion promptly. We presented Pyrrha, a congestion-root-based per-hop flow control. It controls the transmission of flows at a fine granularity without congestion spreading, requiring a minimum number of queues. The performance of flows can be significantly improved. We are currently discussing with a major vendor the implementation of Pyrrha in its products.

Acknowledgments

We sincerely thank our shepherd Gianni Antichi and the anonymous reviewers for their valuable feedback on this paper. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, 62072228, and 62172204, and the Fundamental Research Funds for the Central Universities.

References

[1] Source routing. http://en.wikipedia.org/wiki/Sourcerouting, 2018.

[2] Hierarchical class of service overview. https://www.juniper.net/documentation/us/en/software/junos/cos/topics/concept/hierarchical-cos-overview.html, 2021.

[3] Introduction to hqos. https://support.huawei.com/hedex/hdx.do?docid=EDOC1100168821&id=EN-US_TASK_0172371452&lang=en, 2022.

[4] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI* (2016), pp. 265–283.

[5] Addanki, V., Apostolaki, M., Ghobadi, M., Schmid, S., and Vanbever, L. Abm: active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 36–52.

[6] Addanki, V., Michel, O., and Schmid, S. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), USENIX Association.

[7] Afek, Y., Kutten, S., and Yung, M. Memory-efficient self stabilizing protocols for general networks. In *International Workshop on Distributed Algorithms* (1990), Springer, pp. 15–28.

[8] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review 38*, 4 (2008), 63–74.

[9] Alcoz, A. G., Dietmüller, A., and Vanbever, L. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 59–76.

[10] Alibaba. HPCC simulator. https://github.com/alibaba-edu/High-Precision-Congestion-Control, 2019.

[11] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center TCP (DCTCP). In *ACM SIGCOMM* (2011).

[12] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM 2013* (2013), vol. 43, ACM, pp. 435–446.

[13] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI* (2010).

[14] Arslan, S., Li, Y., Kumar, G., and Dukkipati, N. Bolt:Sub-RTT congestion control for Ultra-Low latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 219–236.

[15] Arun, V., Alizadeh, M., and Balakrishnan, H. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 177–192.

[16] Association., I. T. Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE), 2014.

[17] Association., I. T. InfiniBandTM Architecture Specification Volume 1 Release 1.4. (2020). https://cw.infinibandta.org/document/dl/8567, 2020.

[18] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS* (2012).

[19] Ben Basat, R., Ramanathan, S., Li, Y., Antichi, G., Yu, M., and Mitzenmacher, M. PINT: Probabilistic In-Band Network Telemetry. In *ACM SIGCOMM* (2020).

[20] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *ACM IMC* (2010).

[21] Benson, T., Anand, A., Akella, A., and Zhang, M. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review 40*, 1 (2010), 92–99.

[22] Broadcom. Bcm56990 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series, 2019.

[23] Broadcom. Bcm88800 traffic management architecture. https://docs.broadcom.com/doc/88800-DG1-PUB, 2021.

[24] Cai, Q., Arashloo, M. T., and Agarwal, R. dcpim: Near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 53–65.

[25] Cho, I., Jang, K., and Han, D. Credit-scheduled delay-bounded congestion control for datacenters. In *ACM SIGCOMM* (2017).

[26] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing data transfers in computer clusters with orchestra.

[27] Cisco. Cisco nexus 5548p switch architecture. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-5548p-switch/white_paper_c11-622479.html, 2010.

[28] Cisco. Cef polarization. https://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/116376-technote-cef-00.html, 2013.

[29] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[30] Fitzpatrick, B. Memcached: a Distributed Memory Object Caching System. http://www.memcached.org/, 2011.

[31] Gao, P. X., Narayan, A., Kumar, G., Agarwal, R., Ratnasamy, S., and Shenker, S. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT* (2015).

[32] Gärtner, F. C. A survey of self-stabilizing spanning-tree construction algorithms.

[33] Ghabashneh, E., Zhao, Y., Lumezanu, C., Spring, N., Sundaresan, S., and Rao, S. A microscopic view of bursts, buffer contention, and loss in data centers. In *Proceedings of the 22nd ACM Internet Measurement Conference* (2022), IMC '22, Association for Computing Machinery.

[34] Ghorbani, S., Yang, Z., Godfrey, P., Ganjali, Y., and Firoozshahian, A. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 225–238.

[35] Goyal, P., Narayan, A., Cangialosi, F., Narayana, S., Alizadeh, M., and Balakrishnan, H. Elasticity detection: A building block for internet congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 158–176.

[36] Goyal, P., Shah, P., Sharma, N. K., Alizadeh, M., and Anderson, T. E. Backpressure flow control. In *NSDI* (2022).

[37] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Vl2: A scalable and flexible data center network. In *ACM SIGCOMM* (2009).

[38] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N., Moore, A. W., Hand, S., and Crowcroft, J. Queues don't matter when you can jump them! In *NSDI* (2015), USENIX.

[39] Group, T. P. A. W. P4$_{16}$ portable switch architecture (psa). https://p4lang.github.io/p4-spec/docs/PSA.pdf, 2021.

[40] Guo, C., Wu, H., Deng, Z., Soni, G., Ye, J., Padhye, J., and Lipshteyn, M. Rdma over commodity ethernet at scale. In *ACM SIGCOMM* (2016).

[41] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM* (2017).

[42] He, J., Zhai, J., Antunes, T., Wang, H., Luo, F., Shi, S., and Li, Q. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *PPoP* (2022).

[43] Hu, S., Bai, W., Zeng, G., Wang, Z., Qiao, B., Chen, K., Tan, K., and Wang, Y. Aeolus: A building block for proactive transport in datacenters. In *ACM SIGCOMM* (2020), ACM.

[44] Hu, S., Chen, K., Wu, H., Bai, W., Lan, C., Wang, H., Zhao, H., and Guo, C. Explicit path control in commodity data centers: Design and applications. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 15–28.

[45] Hu, S., Zhu, Y., Cheng, P., Guo, C., Tan, K., Padhye, J., and Chen, K. Tagger: Practical pfc deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies* (2017), ACM, pp. 451–463.

[46] IEEE. 802.11qbb. priority based flow control. https://1.ieee802.org/dcb/802-1qbb/, 2011.

[47] IEEE. P802.1qcz – congestion isolation. https://1.ieee802.org/tsn/802-1qcz/, 2023.

[48] Intel. Intel tofino2 – a 12.9tbps p4-programmable ethernet switch. https://ieeexplore.ieee.org/document/9220636, 2020.

[49] Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (2024), pp. 745–760.

[50] Joglekar, A., Kounavis, M. E., and Berry, F. L. A scalable and high performance software iscsi implementation. In *FAST* (2005), vol. 5, pp. 20–20.

[51] Juniper. Understanding cos virtual output queues (voqs) on qfx10000 switches. https://www.juniper.net/documentation/en_US/junos/topics/concept/cos-qfx-series-voq-understanding.html, 2017.

[52] Kabbani, A., Vamanan, B., Hasan, J., and Duchene, F. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), pp. 149–160.

[53] KAIST. Expresspass simulator. https://github.com/kaist-ina/ns2-xpass, 2017.

[54] Karol, M., Golestani, S. J., and Lee, D. Prevention of deadlocks and livelocks in lossless backpressured packet networks. *IEEE/ACM Transactions on Networking* (2003).

[55] Kumar, G., Dukkipati, N., Jang, K., Wassel, H., Wu, X., Montazeri, B., Wang, Y., Springborn, K., Alfeld, C., Ryan, M., Wetherall, D., and Vahdat, A. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM* (2020).

[56] Li, Y., Miao, R., Liu, H. H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., et al. Hpcc: High precision congestion control. In *ACM SIGCOMM* (2019).

[57] Lim, H., Bai, W., Zhu, Y., Jung, Y., and Han, D. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021).

[58] Lim, H., Kim, J., Cho, I., Jang, K., Bai, W., and Han, D. Flexpass: A case for flexible credit-based transport for datacenter networks. In *Proceedings of the Eighteenth European Conference on Computer Systems* (2023), pp. 606–622.

[59] Liu, K. Supplymental materials of Pyrrha. https://github.com/NASA-NJU/Pyrrha, 2024.

[60] Liu, K., Tian, C., Wang, Q., Zheng, H., Yu, P., Sun, W., Xu, Y., Meng, K., Han, L., Fu, J., Dou, W., and Chen, G. Floodgate: Taming incast in datacenter networks. CoNEXT '21.

[61] Liu, S., Ghalayini, A., Alizadeh, M., Prabhakar, B., Rosenblum, M., and Sivaraman, A. Breaking the transience-equilibrium nexus: A new approach to datacenter packet transport. In *NSDI* (2021).

[62] Liu, V., Halperin, D., Krishnamurthy, A., and Anderson, T. F10: A fault-tolerant engineered network. In *NSDI 13* (2013), USENIX Association.

[63] Liu, V., Halperin, D., Krishnamurthy, A., and Anderson, T. F10: A Fault-Tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 399–412.

[64] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review* (2008).

[65] Mittal, R., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., Zats, D., et al. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM* (2015).

[66] Mittal, R., Shpiner, A., Panda, A., Zahavi, E., Krishnamurthy, A., Ratnasamy, S., and Shenker, S. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 313–326.

[67] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM* (2018).

[68] Nachiondo, T., Flich, J., and Duato, J. Buffer management strategies to reduce hol blocking. *IEEE Trans. Parallel Distrib. Syst.* (2010).

[69] Pan, R., Prabhakar, B., and Laxmikantha, A. Qcn: Quantized congestion notification. *IEEE802 1* (2007), 52–83.

[70] Peng, Y., Zhu, Y., Chen, Y., Bao, Y., Yi, B., Lan, C., Wu, C., and Guo, C. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 16–29.

[71] Peterson, W. W., and Brown, D. T. Cyclic codes for error detection. *Proceedings of the IRE* (1961).

[72] Qian, K., Xi, Y., Cao, J., Gao, J., Xu, Y., Guan, Y., Fu, B., Shi, X., Zhu, F., Miao, R., Wang, C., Wang, P., Zhang, P., Zeng, X., Ruan, E., Yao, Z., Zhai, E., and Cai, D. Alibaba hpn: A data center network for large language model training. In *SIGCOMM* (2024).

[73] Qureshi, M. A., Cheng, Y., Yin, Q., Fu, Q., Kumar, G., Moshref, M., Yan, J., Jacobson, V., Wetherall, D., and Kabbani, A. PLB: congestion signals are simple and effective for network load balancing. In *ACM SIGCOMM* (2022), pp. 207–218.

[74] RackSolutions. How many servers does a data center have? https://www.racksolutions.com/news/blog/how-many-servers-does-a-data-center-have/, 2020.

[75] Rajasekaran, S., Ghobadi, M., and Akella, A. CASSINI: Network-Aware job scheduling in machine learning clusters. In *NSDI 24* (Santa Clara, CA, 2024), USENIX Association.

[76] Rajasekaran, S., Ghobadi, M., Kumar, G., and Akella, A. Congestion control in machine learning clusters. In *HotNets* (2022).

[77] Ros-Giralt, J., Amsel, N., Yellamraju, S., Ezick, J., Lethin, R., Jiang, Y., Feng, A., and Tassiulas, L. A quantitative theory of bottleneck structures for data networks. *arXiv preprint arXiv:2210.03534* (2022).

[78] Ros-Giralt, J., Amsel, N., Yellamraju, S., Ezick, J., Lethin, R., Jiang, Y., Feng, A., Tassiulas, L., Wu, Z., Teh, M. Y., et al. Designing data center networks using bottleneck structures. In *SIGCOMM* (2021).

[79] Rosen, E., Viswanathan, A., and Callon, R. Rfc3031: Multiprotocol label switching architecture, 2001.

[80] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2015).

[81] Sharma, N. K., Liu, M., Atreya, K., and Krishnamurthy, A. Approximating fair queueing on reconfigurable switches. NSDI'18, USENIX Association.

[82] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM* (2015).

[83] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 44–57.

[84] Sonchack, J., Aviv, A. J., Keller, E., and Smith, J. M. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–16.

[85] W. Dally, P. C., and Dennison, L. Architecture of the avici terabit switch/router. *Proc. of 6th Hot Interconnects* (1998).

[86] Wang, W., Moshref, M., Li, Y., Kumar, G., Ng, T. E., Cardwell, N., and Dukkipati, N. Poseidon: Efficient, robust, and practical datacenter CC via deployable INT. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 255–274.

[87] Xu, Y., He, K., Wang, R., Yu, M., Duffield, N., Wassel, H., Zhang, S., Poutievski, L., Zhou, J., and Vahdat, A. Hashing design in modern networks: Challenges and mitigation techniques. In *ATC* (2022).

[88] Yang, M., Baban, A., Kugel, V., Libby, J., Mackie, S., Kananda, S. S. R., Wu, C.-H., and Ghobadi, M. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 633–648.

[89] Yu, Z., Wu, J., Braverman, V., Stoica, I., and Jin, X. Twenty years after: Hierarchical core-stateless fair queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation* (2021), pp. 29–45.

[90] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI* (2012).

[91] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI* (2012).

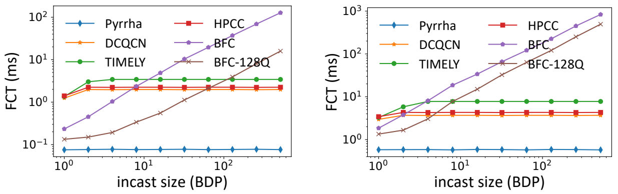
[92] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

[93] Zhang, Z., Zheng, H., Hu, J., Yu, X., Qi, C., Shi, X., and Wang, G. Hashing linearity enables relative path control in data centers. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 855–862.

[94] Zhiyu Zhang, Shili Chen, R. Y. R. S. H. M. H. W. Z. C. G. F. Y. F. W. S. S. L., and Xu, Y. vpifo: Virtualized packet scheduler for programmable hierarchical scheduling in high-speed networks. In *SIGCOMM* (2024).

[95] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion control for large-scale rdma deployments. In *ACM SIGCOMM* (2015).

APPENDIX



(a) Average FCT.

(b) 99th-tail FCT

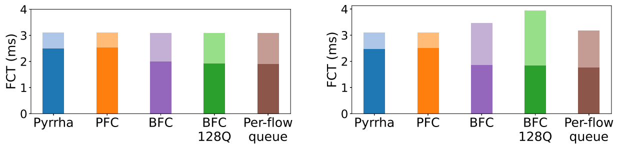
Figure 17: When the size of incast flows varies.

A Discussions

Dealing with link failures. For link failure that exactly occurs at the port of congestion roots, Pyrrha sends RESUME frames to its upstream switches to avoid unnecessary further control. For link failure on other ports, the failure is transparent to Pyrrha when source routing is used. When hash calculation is used to get flows' path, the calculation can be dynamically adjusted by introducing a rerouting table when route changes by leveraging existing rerouting protocols to propagate the rerouting information.

Related works. The IEEE standard group advocates Congestion Isolation (CI) [47], supporting the isolation of congested data flows within datacenter. An egress port can identify flows causing congestion and isolate them locally. The progress of the IEEE standard proved the necessity of Pyrrha from the side. However, CI can only control congested flows that already arrived at the congested port. Besides, CI uses only one dedicated queue for all congested flows, resulting in HOL blocking among congested flows passing through different congested ports.

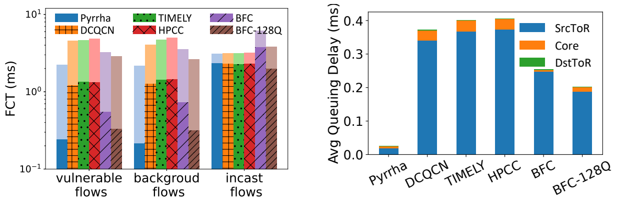
In addition, there are several existing lines of work that go beyond flow control and congestion control. dcPIM [24] takes several RTTs to match senders and receivers before starting transmission to avoid congestion. TCD targets solving the problem that traditional flow control (*e.g.*, PFC and CBFC [17]) interferes with the congestion detection of congestion control protocols. It proposes an undetermined state by leveraging the ON-OFF pattern caused by flow control, and only the flow identified as congested reduces its rate according to congestion control. It can give more penalty to congested flows afterward but does not handle congestion that already occurs. ABM [5] is an efficient buffer-sharing scheme that leverages both spatial and temporal features of the buffer. Queue scheduling approaches target at providing approximate fair queuing [81, 89] or strict priority [12, 38, 9, 83]. Scheduling approaches do not differentiate congested flows from those uncongested thus they can not eliminate HOL blocking. CASSINI [76, 75] adjusts the communication phases of ML jobs to share the bandwidth. These works are orthogonal to Pyrrha.



(a) Web Server (incast flows).

(b) Web Search (incast flows).

Figure 18: Incast flow performance.



(a) Average and 99th-tail FCT.

(b) Queuing time (uncongested flows).

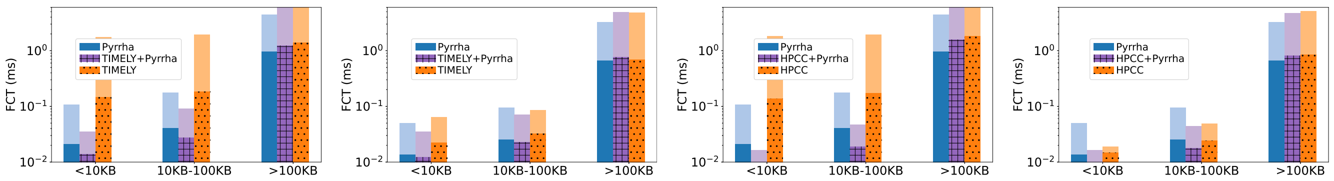
Figure 19: When facing multiple congestion roots.

B Supplemental Results

When the size of incast flows varies. Now we investigate the performance of uncongested flows when the size of incast flows varies. As shown in Figure 17, Pyrrha achieves a stable performance when the size of incast flows increases. This is expected since Pyrrha can already control the transmission of incast flows when the size of incast is smaller than one BDP (§ 6.2). The performance of BFC downgrades to a large extent with long incast flows. Because long incast flows occupy the queues continuously, HOL blocking uncongested flows sharing the same queue with them. For CC protocols, the FCT of uncongested flows becomes stable after the incast size increases to three BDPs since congestion control begins taking effect.

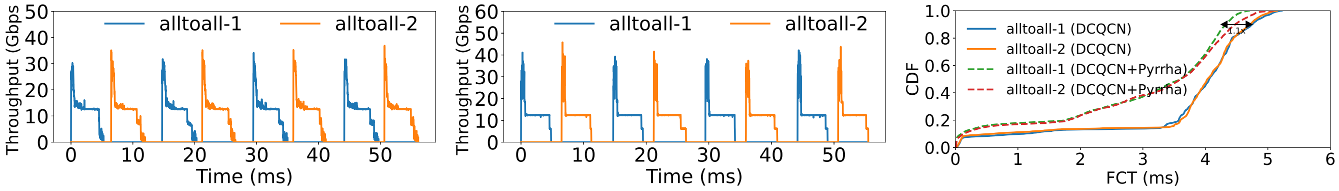
When facing multiple congestion roots. We investigate the scenario where multiple congestion roots exist. An uneven hash function is used to conduct load imbalance. ToR switches can choose among four core switches to forward, the possibility to route to the first core switch is 50%, and the other three paths share the left 50% uniformly. This is called hash polarization, which can occur in datacenter networks. Flows are generated as the incastmix scenario (§ 6.2). Hence, there are multiple congestion roots in networks, *e.g.*, load imbalance on the upstream ports of ToR switches and incast on their downstream ports.

Figure 19 shows the performance of different approaches. Pyrrha reduces the average and tail FCT of non-incast flows. Pyrrha detects the congestion on source ToR switches and sends congestion notifications to the connected source hosts. Then, senders can send other flows that do not pass the imbalanced ports to utilize the network. For congested flows passing through both load imbalance and incast ports, their transmission is well controlled. Hence, the queuing delay



(a) Web Server (vulnerable flows). (b) Web Server (uncongested flows). (c) Web Server (vulnerable flows). (d) Web Server (uncongested flows).

Figure 20: FCT performance under k=16 fat-tree.



(a) Throughput (DCQCN). (b) Throughput (DCQCN+Pyrrha). (c) FCT comparison.

Figure 21: Performance comparison under MoE workloads (interleaved phase).

of Poisson flows is quite small. For CC approaches, they can not alleviate the throughput downgrade caused by load imbalance, hence FCTs of Poisson flows are prolonged.

Supplemental results under k=16 fat-tree. Figure 20 shows the additional evaluations of Pyrrha, TIMELY, and HPCC under the k=16 fat-tree. The results are consistent with Figure 6.3 where Pyrrha is compared with DCQCN.

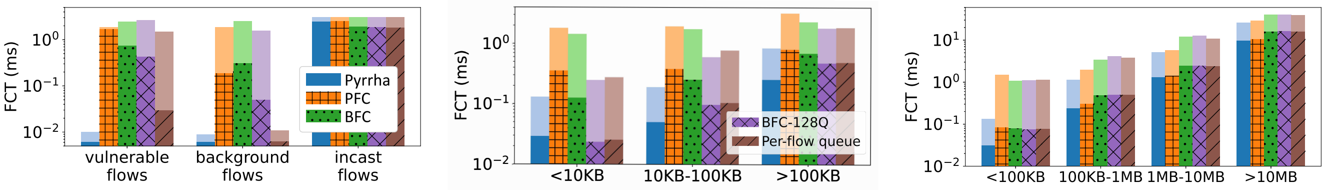
Per-flow queue scheme. Figure 22 shows the performance of flow control protocols in incastmix scenarios, including the per-flow queue scheme. For Memcached, the per-flow queue scheme can achieve as low latency as Pyrrha for background flows. While for vulnerable flows, the per-flow queue scheme can not reduce their tail latencies. Due to the fact that the per-flow queue scheme does not start to control a newly arrived congested flow until it is transmitted to the congested port. It induces a large buffer occupancy at the network bottleneck. Moreover, vulnerable flows and congested flows share the same bottleneck at core switches. When there are a large number of congested flows, vulnerable flows can only get a small fraction of bandwidth share. As a by-product, the per-flow queue scheme reduces the average FCTs of incast flows, as shown in Figure 18.

CC with no window. Figure 26 shows the performance of Pyrrha, DCQCN, and HPCC when no sending window is applied. In comparison to protocols with sending window, the same traffic pattern as Figure 12 is generated. Without the sending window, the queuing delay on source ToR is increased across all protocols, as shown in Figure 26(e). The rationale is the absence of initial transmission control allows for a greater volume of traffic to be injected into the network, leading to longer queue length and increased queuing delays at the source ToRs before CC or FC steps into. DCQCN experiences a larger degree of congestion as each flow can concurrently inject a larger volume of traffic into the network. Likewise, DCQCN+IRN suffers from a larger loss rate which results in performance degradation. In the absence of the sending window, HPCC endures a more severe performance degradation since HPCC is designed to be a window-based protocol. Benefiting from the fast reaction to network congestion, the latency of Pyrrha is slightly increased compared to that with sending window.

Performance under MoE workloads (interleaved phase). Figure 21 depicts the performance of two groups of all-to-all flows with interleaved phase, *i.e.*, the two groups of traffic do not overlap with each other. With the interleaved phase, two groups of flows are slightly affected by each other. By adding Pyrrha to DCQCN, there is still performance improvement of the latency of flows, as depicted in Figure 21(c). This is because the traffic of all-to-all1/all-to-all2 itself can induce transient congestion and Pyrrha can handle it better.

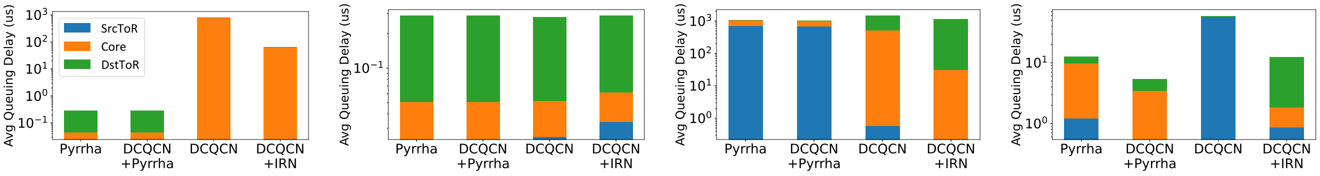
Comparing with TIMELY. We use the parameters recommended by the TIMELY authors in their paper. As shown in Figure 25, when adding Pyrrha on TIMELY, the performance improvement of TIMELY is similar to that for DCQCN.

Deep dive of the per-hop queuing delay. To fully understand how Pyrrha works, we investigate the flows' queuing time among different hops, where the experiments settings is the same as in § 6.2. Figure 23 shows the result. Queuing time is split into three parts, where SrcToR denotes the uplink ports of ToR, *i.e.*, the first hop of packets. Likewise, DstToR denotes the downlink ports of ToR, *i.e.*, the last hop of packets. The average queuing time reflects the extent of HOL blocking that uncongested flows encounter. For DCQCN, vulnerable flows are HOL blocked by incast flows at the core switches. When adding Pyrrha to DCQCN, it eliminates the queuing time for vulnerable flows significantly. For Web Server, Pyrrha also helps reduce the queuing delay of background flows since it does not cause congestion



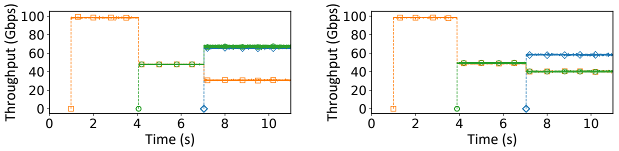
(a) Memcached (overall flows). (b) Web Server (uncongested flows). (c) Web Search (uncongested flows).

Figure 22: FCT performance of FC (per-flow queue included).



(a) Memcached (vulnerable flows). (b) Memcached (background flows). (c) Memcached (incast flows). (d) Web Server (background flows).

Figure 23: Queuing time (DCQCN) is split into three parts, *i.e.*, time spent on SrcToR, core, and DstToR.



(a) Pyrrha (congestion root varies.) (b) PFC (congestion root varies.)

Figure 24: Throughput of testbed experiments (additional).

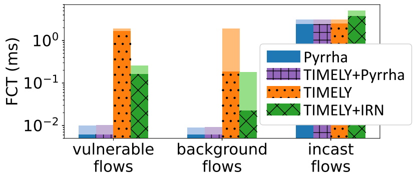


Figure 25: TIMELY: Memcached (overall flows).

spreading, and only flows causing congestion are paused. Pyrrha reacts quickly on congestion at the last hop, hence the queuing time of incast flows is mainly composed of the time spent on the srcToR and core switches.

Pyrrha is robust when congestion roots vary. We provide additional testbed results when congestion roots vary. The testbed setting is the same as in § 5.2. When flows on three hosts S2, VS, and S1 start to arrive at 1s, 4s, and 7s, respectively, the congestion root changes from the core switch to the destination ToR. Pyrrha can also converge quickly and achieve good throughput.

B.1 Parameter and Mechanism Validation

Parameter Selection. In this section, the performance of Pyrrha under different parameters is evaluated. We scan parameters under incastmix scenarios of Memcached. Figure 27 depicts the 99th-tail FCT of the vulnerable flows. K_{resume} is set to 1 when sweeping K_{pause}, and K_{pause} is set to 10 when sweeping K_{resume}. Pyrrha is relatively insensitive to different K_{pause} and K_{resume} value, which is preferred in the industry.

As shown in Figure 27(a), Pyrrha achieves the smallest tail latency when K_{pause} ranges from 2 to 35. When the value of K_{pause} is larger than 35, the tail latency increases. This is because Pyrrha reacts slowly on congestion with a large pause threshold.

As shown in Figure 27(b), the tail latency of Pyrrha remains unchanged when sweeping K_{resume}. For that, with a fixed value of K_{pause}, the timing and correctness of congestion root identification are the same, and K_{resume} only affects the queuing delay arrangement of congested flows on different hops.

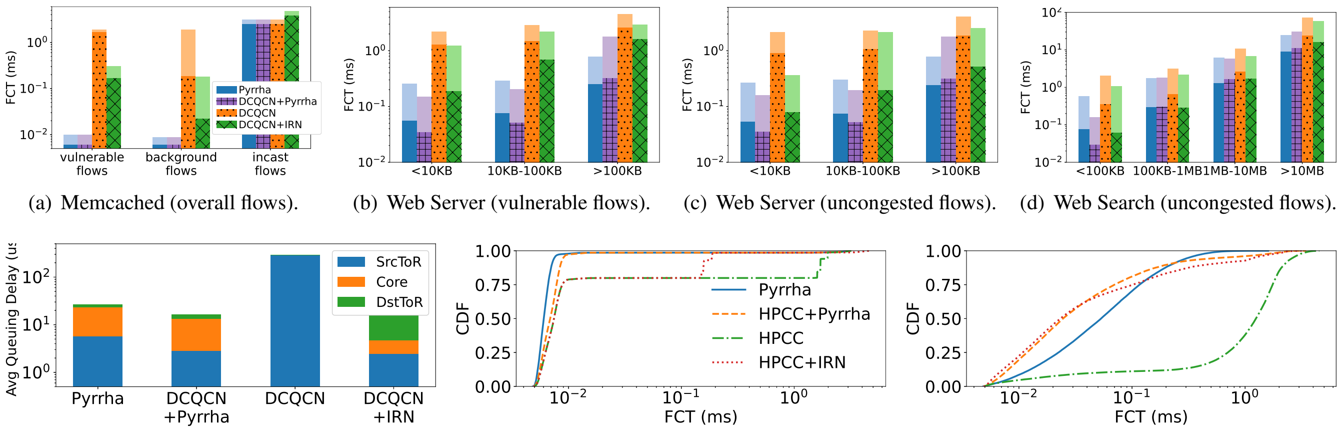
Mechanism validation. With merging, Pyrrha is insensitive to K_{pause} and K_{resume}. As shown in Figure 27(a), without merging, when K_{pause} varies (*e.g.*, 2-35), the inaccurate root detection induces queuing of vulnerable flows. While with merging, those false-positive congestion roots caused by inappropriate parameters could be corrected. The queuing delay shown in Figure 28(a) and 28(b) validates it.

Figure 27(b) demonstrates the performance when K_{resume} varies. Notice that K_{pause} here is set to 10, and the incorrect congestion detection exists. Figure 28(c) shows that the queuing delay increases when K_{resume} is set to a smaller value. This is due to a small resume threshold could be strict to resume and vulnerable flows in the IQ of the false-positive root can be blocked for a long time. With merging, different values of K_{resume} does not influence the queuing delay, as shown in Figure28(d).

C IQ Management

C.1 Methodology

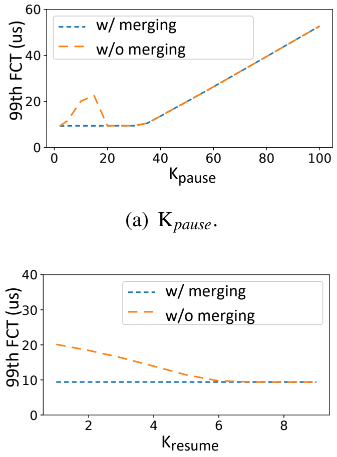
The function of HIQs (*e.g.*, hierarchical congestion matching and in-order delivery) can be achieved by leveraging single-tier IQs. Considering a two-tier HIQ matches congestion roots belonging to two different levels. We denote the single-tier IQ matching two-level congestion roots as IQ_{A_i} and IQ_{B_j}



(a) Memcached (overall flows). (b) Web Server (vulnerable flows). (c) Web Server (uncongested flows). (d) Web Search (uncongested flows).

(e) Queuing time (Web Server background). (f) HPCC: Memcached (overall flows). (g) HPCC: Web Server (uncongested flows).

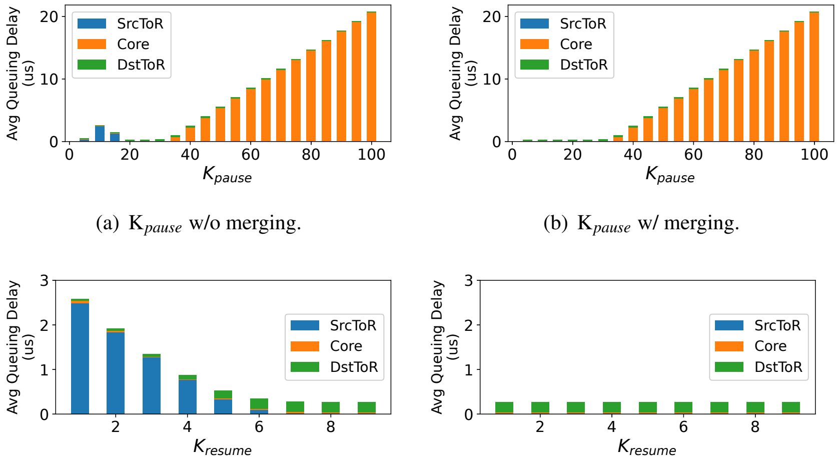
Figure 26: Performance w/o sending window.



(a) K$_{pause}$.

(b) K$_{resume}$.

Figure 27: Parameter selection.



(a) K$_{pause}$ w/o merging. (b) K$_{pause}$ w/ merging.

(c) K$_{resume}$ w/o merging. (d) K$_{resume}$ w/ merging.

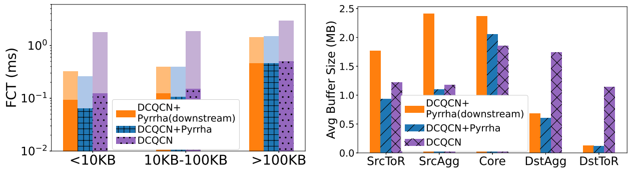
Figure 28: Queuing time variation of vulnerable flows across different value of K$_{pause}$ and K$_{resume}$.

respectively and use lower case letters a_i and b_j to denote the corresponding congestion roots. For a network with two congestion roots a_1 and b_1, there can be flows matching congestion roots a_1, b_1, and $a_1 \wedge b_1$. To handle these three types of congestion flows, for HIQs, IQ$_{A_1}$ and IQ$_{B_1}$ can be arranged into two levels. For single-tier IQs, three dedicated IQs (*i.e.*, IQ$_{A_1}$, IQ$_{B_2}$ and IQ$_{A_1 \wedge B_2}$) are needed. Generally, to isolate different congested flows as in two-tier IQs (*e.g.*, n first level IQs IQ$_{A_1}, \ldots,$IQ$_{A_n}$ and m second level IQs IQ$_{B_1}, \ldots,$IQ$_{B_m}$), the single-tier IQs need additional $n*m$ IQs (*i.e.*, IQ$_{A_1 \wedge B_1}, \ldots$ IQ$_{A_n \wedge B_m}$).

For single-tier IQs, to ensure in-order delivery, it should be careful when a congested flow alters its chosen queue. Scenarios where a flow changes the queue it uses only occur when (i) a congestion root vanishes , or (ii) a new congestion root is detected. For the first scenario, since the IQ to be unassigned must be empty, packets of the flow are not in different queues hence in-order delivery is satisfied naturally. The second scenario is described in § 4.3, where a new congestion root b_1 is detected after a_1. Then IQ$_{B_1}$ and IQ$_{A_1 \wedge B_1}$ are assigned to traffic passing b_1 and traffic passing both congestion roots. Following arrival packets passing through both congestion roots a_1 and b_1 alters from IQ$_{A_1}$ to IQ$_{A_1 \wedge B_1}$. A order mark is inserted into the IQ$_{A_1}$, and IQ$_{A_1 \wedge B_1}$ can not be resumed until the order mark is dequeued from IQ$_{A_1}$. Note that IQ$_{A_1}$ can be paused by congestion root b_1 before the order mark is dequeued due to that flows passing through both port a_1 and b_1 can be pushed into IQ$_{A_1}$ before congestion root b_1 is detected. Given that Pyrrha detects congestion roots rapidly, packets in IQ$_{A_1}$ can be moderate hence the order mark can be dequeued in a short time.

C.2 Experimental Verification

To reduce the complexity, we can analog two-tier HIQs which is enough for two-tier Clos-networks. For three-tier



(a) FCTs.

(b) Buffer Occupancy.

Figure 29: Performance when using single-tier congestion queue to perform a two-tier HIQ.

Clos or fat-tree networks, it only matches congestion roots that two hops away from the destination host. And we leverage CC protocols to handle the congestion close to the source host.

We evaluate the performance of Pyrrha when single-tier congestion queues are used to analog the behavior of a two-tier HIQ in a three-tier topology. Figure 29 shows the performance of Pyrrha and Pyrrha handling downstream congestion roots by leveraging single-tier IQs. Results show that when comparing Pyrrha (downstream)+DCQCN with Pyrrha+DCQCN, the latency of flows smaller than 100KB is increased. But Pyrrha (downstream)+DCQCN reduces the latency of flows significantly compared to pure DCQCN.

D Pyrrha Prototype and Discussion

In this section, we present our Tofino2 prototype, with Figure 30 depicting its pipeline. We divide the pipeline into three parts, each detailed in a separate subsection (Appendix D.1 - D.3). Finally, we summarize the Tofino2 features leveraged in our prototype and examine the practicality of implementing Pyrrha on the RTC (Run-To-Completion) architecture, *e.g.*, Juniper Trio [88].

D.1 Ingress Pipeline

Packet Classifier. The packet classifier is located at the front of the ingress pipeline. The module determines the packet type according to the Ethertype field in the Ethernet header. (We use Ethertypes that are not used by the IEEE standard to mark different types of packets.) There are three types of packets: data packet, signal packet, and generated packet. Different types of packets then go through different modules.

Forward Engine. Only data packets go through the forwarding engine. The forwarding engine looks up the table deployed in advance and determines the egress port of the data packet. This module is similar to that of most commercial switches.

Congestion Root Matcher. This module contains two submodule: route calculation unit and hierarchical congestion root table. The methodology of route calculating is described in § 4.2, and the optimization to reduce the resource usage is detailed in Appendix E.3. And as described in § 5.1, the hierarchical congestion root table records the congestion roots and their states in a hierarchical manner. When a data packet passes, the module first calculates its route, then matches its route against the congestion root table hierarchically. If the packet matches one or more congestion roots in PAUSE state, the matched congested root id(s) is (are) carried in its metadata, which is stored in PHV and will be used in the following modules. When a signal packet passes, the module first calculates the location of the congestion root it indicates, then modifies the state of the corresponding congestion root in the hierarchical congestion root table.

Queue Manager. This module manages the mapping between the congestion root id and queue id. When a packet passes, the module tries to match the congestion root id it carried in metadata against mapping table. If there is no matched entry, the module assigns a new queue to this congestion root. When a queue is unassigned, the module deletes the corresponding entry.

Queue State Detector. *Ghost* is used in this module to detect the change in queue state. When Ghost is triggered by predefined thresholds (K_{pause} or K_{resume}), a Ghost packet is generated to the ingress pipeline and writes the queue length into a register in this module. When passing through this module, the data packet tries to read the value in the register corresponding to the queue in which it will be queued. Then the data packet compares the queue length it read with the previous queue length stored in another register. If the queue length exceeding K_{pause} or falling below K_{resume} for the first time, a PAUSE/RESUME packet is needed to be sent. In such case, the module marks the packet.

At the end of the ingress pipeline is the signal packet module, which contains three sub-modules corresponding to three types of packet.

Signal Packet Creator. Only data packets will pass through this module. Recalling that in the Queue State Detector module, the data packet is marked if a signal packet is needed to be sent. Then this module will check the marker and uses Pktgen to generate a new packet if the marker is valid. The *new programmable Pktgen trigger* of Tofino2 enables carrying some data plane metadata in the generated packet, which greatly expands programmability.

Signal Packet Constructor. The generated packets will pass through this module. This module will reorganize the metadata and construct the header of the signal packet.

Signal Packet Handler. This module will pause/resume the IQ based on the queue information carried in the signal packet. The dataplane pause/resume is achieved via *AFC*, a new feature of Tofino2. If the signal packet switches some flows from a working queue to another queue, the signal will be modified to a Order Mark and pushed into the origin queue. Otherwise, the signal packet is useless and dropped by the module.

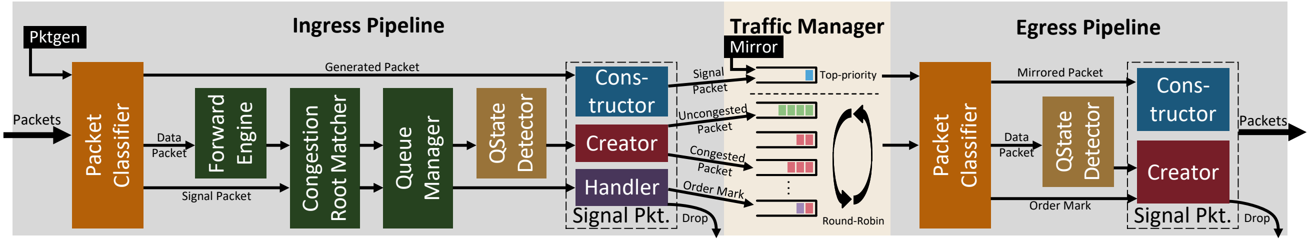


Figure 30: Pipeline of Pyrrha implementation on Tofino2

D.2 Traffic Manager

In the Traffic Manager, we configure a highest priority queue as the queue through which the signal packets pass. Other queues are configured as the same priority and scheduled in a Round-Robin manner. Among these queues, one is used as OQ, and the rest of the queues are used as IQ, which are managed by Queue Manager in Ingress Pipeline.

D.3 Egress Pipeline.

Packet Classifier. The functionality of this module is similar to the Packet Classifier in Ingress Pipeline. All packages are divided into three types: mirror packet, data packet, and Order Mark.

Queue State Detector. Unlike the Queue State Detector in Ingress Pipeline, this module uses the queue length carried by the egress packet to detect the change of queue state. This module mainly focuses on one kind of queue state: CLEAR. If an IQ is clear, it means we can unassign it when other conditions are met.

Signal Packet Creator. Because the programmable Pktgen trigger is not available in Egress Pipeline, this module uses *mirror*, another Tofino feature, to create signal packets. To deliver the signal packet to Ingress Pipeline, the module mirrors the packet to the top-priority queue of the loopback port. For the data packet with a CLEAR mark, this module just mirrors the header to create a small-size signal packet. For the Order Mark, this module mirrors the whole packet and then drops the origin packet.

Signal Packet Constructor. In this module, the mirrored packet is constructed as a signal packet. For the packet mirrored from the data packet's header, the module discards all the origin headers and reconstructs the signal header. For the packet mirrored from the Order Mark, the module only makes minor modifications.

D.4 Feasibility on RTC switches

In our Tofino2 prototype, we utilized the following features that may not be supported by all programmable switches: Ghost thread, packet trigger, and AFC (Advanced Flow Control). (i) Ghost thread is used to get queue lengths at the ingress pipeline. (ii) The packet trigger functionality can generate new packets at the end of both the ingress and egress pipelines. (iii) AFC enables the pausing and resuming of queues in the programmable data plane. We then discuss how to implement these features on other types of programmable switches that adopt RTC (Run-To-Completion) architecture, *e.g.*, Juniper Trio [88].

(i) In programmable switches, queue lengths are typically visible after packets pass through TM (Traffic Manager), *i.e.*, during the process of egress. For Tofino, the Ghost thread is required due to the hardware limitation that information can not be directly conveyed from to egress to ingress. However, in the RTC architecture, the memory is shared between ingress and egress, thus Ghost thread is naturally not required.

(ii) The packet trigger functionality is not directly supported by current RTC switches. Fortunately, through private discussion with a vendor of the RTC switch chips, we have learned that the feature can be easily implemented using background threads and will be integrated into its forthcoming RTC architecture switch chip. More precisely, the design involves dedicated background threads and associated data structures that are reserved for packet generation. These background threads are continuously active, polling the data structures for the notifications of packet generation. When a need arises to generate new packets, the necessary packet generation information is written into the corresponding data structure. Upon detection of this information, the background threads initiate the generation process accordingly.

(iii) Currently, AFC is not supported by existing RTC chips. The support for AFC in Tofino2 demonstrates the feasibility of exposing this functionality to the programmable data plane. Considering the necessity to support prevalent flow control protocols such as PFC, the capability to pause and resume queues should be a standard functionality in RTC. We are optimistic that future releases of RTC switch chips will support this essential functionality.

E Resource Usage on Tofino2

In this section, we detail the resource overhead of the Pyrrha prototype on Tofino2. Tofino2 adopts pipeline architecture, wherein the overhead of storage and computational resources is determined at compile time. It enables us to ascertain Pyrrha's resource requirements without running it in a large-scale cluster. We separately compile three key data structures that are related to the size of the topology to report their overhead across different topological scales (Appendix E.1-E.3). Subsequently, we present the resource usage of processing logic, which remains constant irrespective of the scale of topology (Appendix E.4). Ultimately, we analyze the computational and storage resource complexities of Pyrrha and provided an exhaustive account of resource utilization under several typical network conditions. (Appendix E.5)

According to the statistics reported in megascale [49], the scale of current data centers can reach up to 10,000 nodes. Therefore, we use a fat-tree topology with 11,664 hosts as a representative case for our analysis. In addition, we investigated the largest scale of topology that Pyrrha can support theoretically, *i.e.*, approximately an order of magnitude larger than the state-of-the-art datacenter networks. The results demonstrate that Pyrrha is capable of scaling to large-scale networks. Our prototype of Pyrrha can support a typical modern data center with over 10,000 hosts. Leveraging the rapidly maturating HIQ technology [94], Pyrrha is capable of scaling up to support topology approximately an order of magnitude larger than the typical modern datacenter networks.

Our overhead analysis focuses on the ToR switches since they should track the most extensive status information for their downstream networks, thereby consuming the largest amount of resources. It is notable that the data plane storage and the forwarding plane buffers are segregated in the Tofino2. Thus Pyrrha's consumption of storage does not impact the forwarding performance.

E.1 Path Calculation

This section details how we implement hash-based path calculation for fat-tree topology in our Tofino2 prototype. The path calculation can be divided into two parts: determining available egress ports and selecting the forwarding port from available ports by hashing. Given the fact that switches in datacenter only support limited types of hash functions (*e.g.*, CRC or XOR) to conduct efficient calculation [87, 93] and use switch-specific hash seeds to avoid hash polarization. Hence, Pyrrha switch only needs to store a bitmap to indicate the type of the hash function of other switches along with their hash seeds (§ E.1.1).

In many widely deployed topologies [8, 37, 62, 82], the characteristics of up-down routing and equal paths can be leveraged to optimize the overhead of storing other switches' forwarding tables. We demonstrate that path calculation can be done within $O(n)$ resource usage where n is the number of hosts in the topology. For a fat-tree with 11,664 hosts, Pyrrha's path calculation occupies 0.44MB SRAM (*i.e.*, 1.8% of Tofino2) (§ E.1.2). If IP addresses are organized regularly as discussed in [8], the storage cost can be further reduced to 0.18% of Tofino2 (§ E.1.3). Otherwise, a Pyrrha switch needs to store forwarding tables on other switches to calculate the path, incurring extra storage overhead.

E.1.1 Hash functions used in the data center

Current switches only provide a limited number of hash functions to prevent complex hashing from impeding forwarding rates [87, 50]. To avoid traffic polarization caused by identical hash functions across different switches, it is a common practice to employ switch-specific seeds to derive switch-specific hash functions [28]. Before performing the hash function, the input data (*e.g.*, IP tuple) is XORed with the switch-specific seed, making the output of identical input data and hash function different among switches. Switches' hash seeds are not hardcoded but accessible through the control plane interface [28]. Therefore, a switch can calculate the hash outcomes of other switches by storing a limited number of hash functions and other switches' hash seeds.

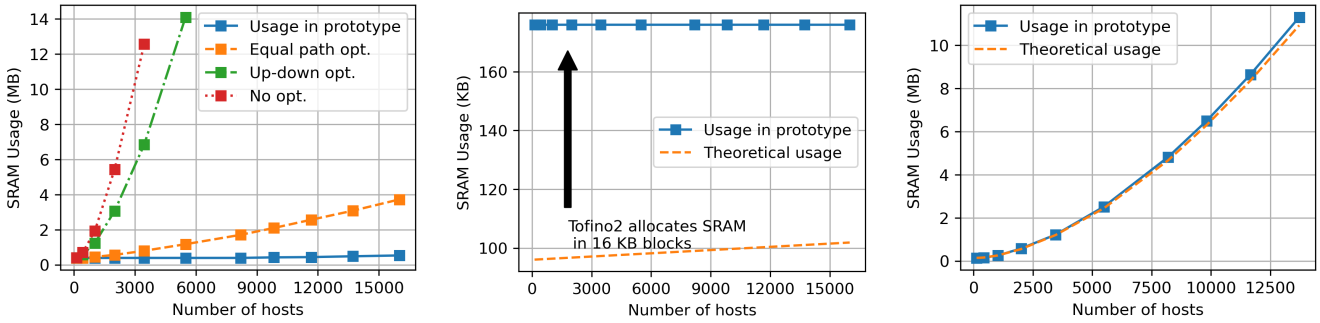
E.1.2 Path Calculation without regular addressing.

To calculate the path, Pyrrha switch needs to store the forwarding tables of other switches. In this subsection, we mainly focus on the structure and the storage overhead of forwarding tables in fat-tree topology. As shown in Figure 31(a), under two topology-relevant optimization, Pyrrha switch can calculate path in $k = 36$ (*i.e.*, 11,664 hosts) fat-tree with 0.44 MB of SRAM. The SRAM usage is acceptable in Tofino2, which has 25 MB SRAM in total. These optimizations can also be applied to other clos-based topologies.

No optimization. Without optimization, a switch needs to store all forwarding tables on other switches. Take fat-tree topology as an example; the size of the forwarding table on a switch is of the order $O(n)$, where n is the number of hosts. The number of the switch is of the order $O(n^{2/3})$. Thus, the size of extra storage usage to calculate the path is of the order $O(n^{5/3})$. Storing all forwarding tables will occupy 12.6 MB in the fat tree with a host number of 3456 (k=24).

Up-down routing optimization. In many topologies, routing adheres to the up-down principle, where packets are first forwarded upward and then downward [45]. In such topologies, switches only need to store upstream routing entries for upward hops and downstream routing entries for downward hops. In a fat-tree, a switch must store entries for all switches in its source pod, all entries for core-layer switches, and only downstream entries for switches in other $(k-1)$ pods. Though it can not reduce the complexity of the resource usage, this optimization can save storage resources on the order of $O(n^{5/3})$ in the fat tree. Storing forwarding tables will occupy 14.1 MB in the fat tree with a host number of 5,488 (k=28) under up-down routing optimization.

Equal path optimization. In many topologies, there are



(a) Forwarding table in path calculation.

(b) Hierarchical congestion root table.

(c) Queue assignment module.

Figure 31: The SRAM usage of three data structures.

multiple equal paths between nodes. Take the source aggregation switch as an example. When a packet destined for another pod arrives, the switch only needs to forward it to any one of the upward links without differentiating between the various upward links. Therefore, the edge switch only needs to store the forwarding table entries of the hosts within their own pod to determine the forwarding decisions on aggregation switches within that pod. Similarly, edge switches only need to store one routing table that maps different IPs to different pods for all core switches, as core switches are only concerned with which pod the packet needs to be forwarded to. For the aggregate switches of the other $k-1$ pods, the edge switch only needs to store $k-1$ different forwarding tables. This optimization can reduce the storage resources usage to the order of $O(n^{4/3})$. Storing forwarding tables will occupy 2.56 MB in the fat tree with a host number of 11,664 (k=36) under equal path optimization.

Full optimization (used in our prototype). Combining the two optimizations mentioned above, we can further reduce the storage resource overhead to the order of $O(n)$. The storage usage of storing forwarding tables is reduced to 0.44 MB in the fat tree with a host number of 11,664 (k=36) under full optimization. In addition, we investigated the largest scale of topology that Pyrrha can support theoretically, *i.e.*, approximately an order of magnitude larger than the state-of-the-art datacenter networks. In our prototype, storing forwarding tables will occupy 8.7 MB in the fat tree with a host number of 524288 (k=128) under full optimization.

Next, we introduce the specific content of the forwarding tables to storage and then analyze the complexity. (i) A ToR switch is required to store $K/2$ downstream entries for itself. If the destination IP of a packet is missed in the downstream entries, the switch should select a port from upward ports by hash to forward the packet. The complexity is of $O(n^{1/3})$. (ii) The ToR should store store $k^2/4$ downstream entries, for all aggregation switches within the same pod. If the destination IP is not within the pod, the aggregation should select an upward port by hash to forward the packet. The complexity is of $O(n^{2/3})$. (iii) The ToR should store $k^3/4$ entries for all core switches, containing the mappings between destination IPs and their corresponding pods, with a complexity of $O(n)$. (iv) Additionally, for the aggregation switches in each of the other $k-1$ pods, $k-1$ tables need to be stored, each with a size of $k^2/4$, to store the mappings of different destination IPs within that pod to the appropriate edge switches, with a complexity of $O(n)$. (v) For all other edge switches, tables of size $k/2$ that record the mappings between destination IPs and hosts should be stored, also with a complexity of $O(n)$. Therefore, under full optimization, the storage overhead complexity for maintaining the forwarding tables is $O(n)$.

Apply optimizations on other topologies. The two optimizations mentioned above can be applied to most clos-based topologies [37, 63, 82].

E.1.3 Path Calculation with regular addressing

In this subsection, we discuss how to conserve storage overhead for forwarding tables based on the regularity of the topology's addressing. Our discussion is based on the fat-tree topology with addressing method presented in [8]. We believe that this approach can be applicable to the vast majority of topologies with regular addressing schemes.

IP address format. The address of a host is $10.pod.switch.host$, where *pod* is the pod number, *switch* is the edge switch's position in the pod (counting from left to right) the host connected to, and *host* is the host's position among hosts connected to the switch.

Switch ID and port ID format. The result of path calculation is the switch ID and egress port ID that a packet traverses at each hop. The switch ID and port ID format given below supports fat-tree topology with k up to 128. Aligned with IP addresses, IDs are counted from left to right.

The switch ID is 16 bits in length. For edge and aggregation switches, the upper 8 bits represent the pod number, while the lower 8 bits denote the switch's position among the

Algorithm 1: Path Calculation In Fat-tree

```
   Input: $tuple$: IP-tuple of the incoming packet.
1  Procedure PathCalculation($tuple$)
2      if $tuple.dstIP.pod == tuple.srcIP.pod$ then
3          if $tuple.dstIP.switch == tuple.srcIP.switch$ then
4              $edgeLocal \leftarrow true$
5          else
6              $aggLocal \leftarrow true$
7      if $edge\_local == false$ then
8          $edgePortID \leftarrow \{1, \mathtt{Hash}(tuple, localSeed)\}$
9          if $agg\_local == false$ then
10             $aggSeed \leftarrow \mathtt{LookupSeed}(edgePortID)$
11             $aggPortID \leftarrow \{1, \mathtt{Hash}(tuple, aggSeed)\}$
```

Source edge switch	SwitchID	It is already known
	PortID	*edgePortID*
Source aggregation switch	SwitchID	$\{srcIP.pod, 0, edgePortID[6:0]\}$
	PortID	*aggPortID*
Core switch	SwitchID	$\{0, edgePortID[6:0], 0, aggPortID[6:0]\}$
	PortID	$\{dstIP.pod\}$
Destination aggregation switch	SwitchID	$\{dstIP.pod, 0, edgePortID[6:0]\}$
	PortID	$\{1, dstIP.switch\}$
Destination edge switch	SwitchID	$\{dstIP.pod, 1, dstIP.switch\}$
	PortID	$\{dstIP.host\}$

Table 1: Obtain switch ID and port ID from existing data.

same layer switches in the pod. For core switches, the upper 8 bits indicate the position of the connected aggregation switch (noting that a core switch connects to aggregation switches at the same position across different pods). The lower 8 bits are the switch's position among core switches connected to the same aggregation switch. Switch IDs may be identical across different layers by design to conserve encoding space and reduce computational overhead. This does not cause issues since Pyrrha's congestion root table is hierarchical (§ 5.1).

The port ID is 8 bits in length. Edge switches and aggregation switches' ports can be categorized as uplink and downlink ports. The highest bit of the uplink port ID is 1, while it is 0 for a downlink port. The remaining 7 bits are the port's position within its respective uplink or downlink category. For core switches, the port ID is the connected pod number.

Path Calculation. After careful optimization, the path calculation in fat-tree only needs up to one lookup and two hash functions. We next explain how to calculate path hop-by-hop on the source edge switch. The pseudocode for path calculation is shown in Algorithm 1. Note that the logic of the P4 code is Match-Action Table (MAT) based, and the pseudocode only presents the calculation logic. After two rounds of hashing, the path for each hop can be obtained using the method described in Table 1. In both the algorithm and the tables, curly braces $\{\}$ denote the bit concatenation, and square brackets $[\,]$ denote the bit slice, which can be easily implemented by setting the key of the MAT. In the concatenation, the 0 and 1 are both 1 bit in width.

(i) Path at source edge switch. The source edge switch is aware of its own switch ID. When calculating the port ID, it is necessary to first determine whether the destination is under the same source edge switch (line 2-4). If so, there is no need to perform any calculations. Instead, the switch simply considers itself as the destination edge switch and uses $\{dstIP.host\}$ as the port ID. If not, it calculates the port ID via hashing IP-tuple and its own hash seed (line 8).

(ii) Path at source aggregation switch. The source aggregation switch id is $\{srcIP.pod, 0, edgePortID[6:0]\}$ as $edgePortID[6:0]$ is both the edge switch's up port position and the aggregation switch's position according to fat-tree's scheme. Similar to the prior case, Pyrrha needs to determine whether the destination is in the same pod (line 2-6). If so, no calculation is needed and this aggregation switch is considered as a destination aggregation switch in the following logic. If not, Pyrrha will perform the hash function. Before calculating the source aggregation switch's hash output, the source edge switch needs to look up the hash seed (line 10). The hash seed table does not need to store the hash seeds for all switches; for an edge switch, it only needs to store the hash seeds of the aggregation switches within the same pod.

(iii) Path at core switch. The switch ID of the core switch the packet will pass by is $\{0, edgePortID[6:0], 0, aggPortID[6:0]\}$. The upper 8 bits are the position of the aggregation switch it connected to and the lower 8 bits are the position of the aggregation switch port it connected to. The egress port ID at the core switch is the destination pod ID, which is available in the IP-tuple.

(iv) Path at destination aggregation switch. The switch ID of the destination aggregation switch the packet will pass by is $\{dstIP.pod, 0, edgePortID[6:0]\}$. The switch position (*i.e.*, the lower 8 bits) is the same as that of the source aggregation switch, eliminating the need for calculation. The port ID at the destination aggregation switch can be obtained from the *switch* field of the destination IP address.

(v) Path at destination edge switch. The switch ID and port ID can be directly obtained from the destination IP address.

Resource usage. Table 2 shows the resource usage of path calculation with regular addressing on the edge switch. Pyrrha use only a slight amount (0.18%) of SRAM to store hash seeds and only two hash function calls. The hash function is built-in in Tofino2, thus do not occupy any extra resource. Among those resources, the most utilized resource is the TableID, which only occupies 2.5% of the total TableID resources available in the Tofino2.

E.2 Hierarchical Congestion Root Table

We store the downstream congestion snapshot in a hierarchical congestion root table. The k^{th} table records ports that

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	40	1.56%
	Ternary Match Input xbar	0	0.00%
	VLIW Instructions	7	1.09%
	Hash Bits	60	0.77%
	Hash Calls	2	1.67%
	Stateful ALU	0	0.00%
	Logical TableID	8	2.50%
Stor.	SRAM	5	0.31%
	TCAM	0	0.00%

Table 2: Resource usage of path calculation with regular addressing.

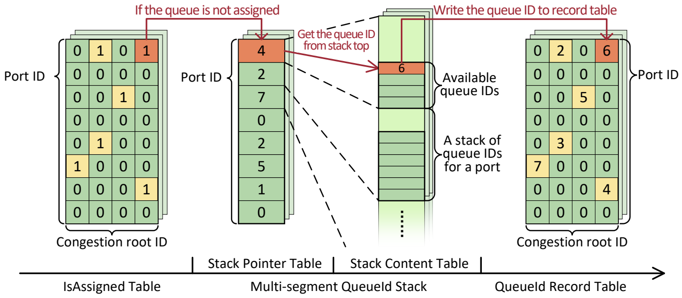


Figure 32: The components of queue assignment module with the process of queue assigning depicted.

are k hop away from the switch, whose keys are <switch-id, port-id>, and the value is a bit, denoting its congestion status. As shown in Figure 31(b), the storage usage of the congestion status snapshot is of the order $O(p)$ where p is the number of ports in the network (*i.e.*, $O(n)$ where n is the number of hosts for most topology). When the number of hosts is small, the usage of SRAM does not increase with the number of hosts. This is because Tofino2 allocates SRAM in blocks, and when the number of hosts is low, the size of each data structure fits within a single block. In our prototype, the hierarchical congestion root table occupies 176 KB in a fat tree with a host number of 11,664 (k=36) and 0.44 MB for 524,288 host (k=128).

E.3 Queue Assignment Module

The queue assignment employs a lazy update design to conserve queue resources. The components of the dynamic queue assignment module and the procedure of queue assignment are shown in Figure 32. The queue allocation module introduced in this section is applicable to both single-layer IQs and HIQs. For HIQs, the queue allocation module is also hierarchical, with each layer of queues being allocated by a corresponding layer of the queue allocation module.

Before entering queue assignment module, a packet is first processed by a congestion root matching module to determine whether it will pass through congestion roots (Appendix D.1). If it will, the packet will carry the IDs of all the congestion roots it will pass and proceed to the queue assignment module.

The packet first passes through *IsAssigned Table*, which has two dimensions: port ID and congestion root ID. This table stores bits to indicate whether a queue has been assigned to a congestion root on a port. Note that in our Tofino2 prototype, we use single-tier IQs to emulate two-level HIQs (§ C). Therefore, the congestion root ID is a concatenation of the IDs of the two potential congestion roots in two layers. If the corresponding bit in the table is 0, it indicates that the corresponding IQ has not been assigned. The packet then modifies this bit to 1 and enters the Multi-segment QueueID Stack to attempt to obtain a queue ID.

The *Multi-segment QueueID Stack* utilizes only two tables, *Stack Pointer Table* and *Stack Content Table*, yet it can independently assign queues in each port. The Stack Pointer Table is indexed by port ID and stores the top pointer of the queue ID stack corresponding to each port. The Stack Content Table holds the remaining available queue IDs for each port. When queue assignment is required, the packet first accesses the Stack Pointer Table to obtain the stack top pointer and then increments the top pointer in the table by 1. Subsequently, it reads the available queue ID from the top of the Stack Content Table. Finally, this queue ID is written into the *QueueID Record Table*, which has the same indexing as the IsAssigned table and stores the queue IDs that have already been assigned.

The pipeline architecture of Tofino2 ensures a strict order in table access among packets. After queue assignment is completed, subsequent packets that pass through the same port and congestion roots will find the corresponding bit in the IsAssigned table set to 1. Then, they will not perform any operations when passing through the Multi-segment QueueID stack. Instead, they will directly read the queue ID from the QueueID Record Table.

When unassigning a queue, the operations are basically the reverse of queue assignment. When an IQ is clear and the IQ is the leaf of the congestion tree, a signal packet is created, which carries the IQ ID to be unassigned as well as the port ID and congestion root ID. It sets the corresponding bit in the IsAssigned table to 0. Then, it increments the stack top pointer in the Stack Pointer Table by 1 and writes the unassigned IQ ID into the new stack top in the Stack Content table. Finally, it clears the corresponding record in the QueueID Record Table.

The SRAM usage of the queue assignment module is shown in Figure 31(c). The resource usage of the queue assignment module mainly stems from the IsAssigned table and the QueueId Record table. When using single-tier IQs, the congestion root ID is a concatenation of the IDs of two layers of congestion roots, which results in a space complexity of $O(n^{5/3})$ for these two tables where n is the

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	89	3.48%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	45	7.03%
	Hash Bits	684	8.22%
	Hash Calls	6	5.00%
	Stateful ALU	14	17.50%
	Logical TableID	63	19.59%
Stor.	SRAM	120	7.50%
	TCAM	31	6.46%

Table 3: Resource usage of Pyrrha's precessing logic.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	277	10.82%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	123	19.22%
	Hash Bits	1647	19.80%
	Hash Calls	36	30.00%
	Stateful ALU	61	76.25%
	Logical TableID	124	38.75%
Stor.	SRAM	712	44.50%
	TCAM	31	6.46%

Table 4: Resource usage of Pyrrha prototype under 11,664 hosts topologies with irregular addressing.

number of hosts in the topology. The SRAM usage of the queue assignment module is shown in Figure 31(c). Queue assignment module for single-tier IQs will occupy 8.64 MB in the fat tree with a host number of 11,664 (k=36), which is adequate to support the scale of typical modern topology.

Leveraging the rapidly maturating HIQ technology [94], on Pyrrha switch that support HIQ, the IsAssigned table and QueueId Record table can be decomposed hierarchically as in Appendix E.2, thereby reducing the space complexity to $O(n)$. We also implement a queue assignment module suitable for HIQ on Tofino2 to investigat the largest scale of topology that Pyrrha can support theoretically, the result shows only 2.2MB of SRAM is required to support up to 524,288 hosts.

E.4 Resource Usage of Processing Logic

In addition to the above three data structures, whose resource usage varies with the network scale, our prototype also includes some processing logic with fixed resource usage. This includes queue length detection, signal packet creating, constructing, handling, as well as queue pausing and resuming (Appendix D). The resource overhead of this processing logic is shown in Table 3. The results indicate that in our prototype, Pyrrha's resource usage is moderate and acceptable for Tofino2.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	233	9.10%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	98	15.31%
	Hash Bits	1251	15.04%
	Hash Calls	33	27.50%
	Stateful ALU	44	55.00%
	Logical TableID	104	32.50%
Stor.	SRAM	294	18.38%
	TCAM	31	6.46%

Table 5: Resource usage of Pyrrha with HIQs under 524,288 hosts topologies with regular addressing.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	250	9.77%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	89	13.91%
	Hash Bits	1426	17.14%
	Hash Calls	29	24.17%
	Stateful ALU	35	43.75%
	Logical TableID	98	30.62%
Stor.	SRAM	450	28.13%
	TCAM	31	6.46%

Table 6: Resource usage of Pyrrha with HIQs under 221,184 hosts topologies with irregular addressing.

E.5 Overall Resource Usage

In this subsection, we analyze the complexity of the overall resource utilization of Pyrrha on the Tofino2 prototype. Subsequently, we present Pyrrha's detailed resource usage across several representative network conditions. According to the statistics reported in megascale [49], the scale of current data centers can reach up to 10,000 nodes. Therefore, we use a fat-tree topology with 11,664 hosts as a representative case for our analysis. In addition, we investigated the largest scale of topology (*i.e.*, a k = 128 fat-tree with 524,288 hosts) that Pyrrha can support theoretically.

Complexity analysis. The computational resources utilized by Pyrrha are predominantly determined by the on the maximum number of network hops, with a minor correlation to the network's scale. Regarding storage resources, the overhead of Pyrrha's data structures is moderate. Denote the number of hosts in the topology as n. With irregular addressing, the storage overhead can be reduced to an order of $O(n)$ through two optimizations. With regular addressing, the storage cost for path calculation is negligible. The storage overhead of a hierarchical congestion root table is also on the order of $O(n)$. In our prototype, the storage cost for the queue assignment module is $O(n^{5/3})$. On switches that support HIQs, the storage cost for the queue allocation module is $O(n)$.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	297	11.60%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	108	16.87%
	Hash Bits	1799	21.62%
	Hash Calls	33	27.50%
	Stateful ALU	44	55.00%
	Logical TableID	112	35.00%
Stor.	SRAM	845	52.81%
	TCAM	31	6.46%

Table 7: Resource usage of Pyrrha with HIQs under 524,288 hosts topologies with irregular addressing.

Overall, the computational resource overhead of Pyrrha remains constant across most network topologies. In our prototype, the storage overhead for Pyrrha is $O(n^{5/3})$. On switches that support HIQ, the storage overhead is $O(n)$.

Detailed usage. We present detailed reports of resource usage for Pyrrha on the Tofino2 prototype across several typical network environments. We first report the resource usage of Tofino2 prototype under the scale of a typical modern data center [49], which is shown in Table 4. In a network with 11,664 hosts (*i.e.*, fat-tree with k=36), the Stateful ALU and Logical TableID are the most utilized computational resources, accounting for 76.25% and 38.75% of the capacity, respectively. And SRAM is the most utilized storage resource, occupying 44.5% of the availale capacity.

In addition, we investigated the largest scale of topology that Pyrrha can support theoretically. Table 5 illustrates the resource usage of Pyrrha on a switch supporting HIQ within a 524k hosts network (*i.e.*, fat-tree with k=128) with regular addressing. The Stateful ALU and Logical TableID are the most utilized computational resources, accounting for 55% and 32.5% of the capacity, respectively. The storage usage is merely 18.38%. Table 6 and 7 shows Pyrrha 's resource usage on a switch supporting HIQ and the topology has 221k and 524k irregular addressed hosts (*i.e.*, fat-tree with k=96 and 128), respectively. In the case of irregular addressing, the computational overhead remains nearly identical, while the SRAM usage increases. In a network with 221k hosts, the SRAM overhead is 28.13%, and in a network with 524k hosts, the SRAM overhead rises to 52.81%, demonstrating that the storage overhead is linear.